UNIVERSITY OF TWENTE.

ORTEC
OPTIMIZE YOUR WORLD

Master's thesis

Automated tuning of an algorithm for the vehicle routing problem

Author:
Koen Henk-Johan Demkes

Supervisors:
Dr. ir. M.R.K. Mes
Dr. ir. J.M.J. Schutten
Prof. dr. J.A. dos Santos Gromicho

October 31, 2014

Automated tuning of an algorithm for the vehicle routing problem

K.H. (Koen) Demkes

Industrial Engineering and Management
Production and Logistic Management

Graduation committee:
Dr. ir. M.R.K. Mes
Dr. ir. J.M.J. Schutten
Prof. dr. J.A. dos Santos Gromicho

University of Twente
Drienerlolaan 5
7522 NB Enschede
The Netherlands
http://www.utwente.edu/

ORTEC
Houtsingel 5
2719 EA Zoetermeer
The Netherlands
http://www.ortec.com/

Management summary

ORTEC develops advanced planning systems for different areas, including vehicle routing. The routing algorithms included in these systems are highly customizable: the customization abilities range from parameter tuning to the definition of the algorithm in terms of composition of steps. Customization is done by means of a configuration: a sequence of algorithms, including the parameters of each algorithm, used to solve the vehicle routing problem. The sequence can be adapted to make it suitable for the customers' problems (i.e., a sequence that is likely to solve the customer's problems well). Within each of these algorithms, we find parameters to control the behavior of the specific algorithm. These parameters can also be varied to make the configuration even more suitable for a certain customer. ORTEC knows from experience that adequately tuning of the configuration dramatically changes the behavior of the algorithms in favor of the specific situation at each customer. The configurations found with the tuning method used in this research perform up to 10% better than the default configuration. The tuning method is also able to find configurations that perform better than configurations tuned by experts of ORTEC. Moreover, expensive time of experts can be saved by using the tuning method.

Problem Tuning the configuration is currently done by the implementation consultants and developers of ORTEC. However, this is an undesired situation as it is a very time (and hence money) consuming task as the environments differ strongly between the customers. Therefore, ORTEC wants to automate the process of tuning these configurations. The main objective of this research is therefore to design and implement a method able to automatically tune the configuration (i.e., configure the algorithms) used for routing, to optimize its behavior on future instances. The time the software needs to solve the problem using the chosen configuration, should be kept within certain bounds. As a consequence, we develop a tuning method that runs offline: not while the software solves problems but during moments at which an abundance of computational resources is available (e.g., weekends). We tune the configuration based on a set of instances from a recent period. By this, the tuning method finds a configuration that is likely to perform well for problem instances in the near future.

Method We develop a tuning method that is based on Sequential Model-Based Algorithm Configuration (SMAC) (Hutter, Hoos, & Leyton-Brown, 2011). The most important aspect of our tuning method is that it allows to predict the performance of a configuration without measuring it. These predictions can be made much faster than actual measurements (seconds versus minutes). We are therefore able to reject configurations with a bad prediction. Hence, we only measure promising configurations. Many state-of-the-art tuning methods are not able to deal with categorical parameters: parameters where the distance between two values cannot be used as a metric. However, such parameters often occur in practical cases. Our tuning method can deal with those categorical parameters.

Results To validate our tuning method, we use a set of practical VRP instances from ORTEC's customers. Our tuning method is able to find a configuration that results in the best performance on such sets, compared to the tuning methods that we consider in this research, namely uRace (van Dijk, 2014) and random sampling. Moreover, the configurations found by our tuning method outperform all other tested configurations, including those tuned by experts. To summarize, our tuning method is cheaper to use than the other tuning methods and results in configurations that outperform any other tested configuration. We also show that the application of our tuning method is not limited to the software for VRP problems, but that it can be used in combination with many other products at ORTEC.

Implications Our tuning method allows ORTEC to improve the service it offers their customers, since the improved configurations lead to improved solutions of the customers' problem instances. Moreover, our tuning method simplifies the process of tuning configurations for ORTEC. Since our tuning method requires almost only computational resources, expensive time of experts is saved.

Preface

This thesis is the result of a project that I enjoyed very much. In this project I was able to combine two of the most interesting subjects of my study Industrial Engineering and Management at the University of Twente: logistics and IT. I am very grateful to ORTEC for giving me this opportunity. I am also very pleased with the achieved results. I believe ORTEC can improve its products based on my findings, delivering even better products to its customers.

This project would not have been possible without the support of many people. Although I cannot mention everyone explicitly, I like to thank some people in particular.

Martijn and Marco, I know you spent a lot of time on reading my drafts, but the constructive feedback, interesting suggestions, and corrected sentences, grammar and spelling resulting from this definitely improved the quality of my thesis. Thank you very much!

Joaquim, thank you for giving me this opportunity, your supervision, critical questions, and genuine interest in my research. Moreover, without your efforts to support the technical implementation of my method, this project would not have become such a success. Tim, thank you for sharing your thoughts about my research, and helping me in setting up an experiment to compare our methods. Colleagues at ORTEC, thank you for your fast and clear answers to my questions, this saved me a lot of time!

Koen

Contents

Chapter 1

Introduction

This research takes place at ORTEC, one of the largest providers of advanced planning and optimization services. ORTEC's software products support many types of problems, including, for example, fleet routing and dispatch, vehicle and pallet loading, workforce scheduling, delivery forecasting, logistics network planning, and warehouse controlling. In this research, we focus on the solution for routing and dispatch: ORTEC Routing and Dispatch (ORD). This product supports the process of distributing goods to customers with a fleet of vehicles. The objective is to minimize the cost of distributing the goods, while taking into account restrictions and company goals. Instead of delivering goods, also picking up goods or a mix of delivering and picking up, can be handled by ORD.

ORD bundles the given tasks into routes and adds the right resources (trucks and drivers). This task is handled by an optimizer, of which two exist for ORD: COMTEC[1] Distribution Planner Service (CDPS) and COMTEC Vehicle Routing Service (CVRS). Although they provide basically the same functionality, differences exist. CDPS is mainly used for solving problems that have a very large number of orders (i.e., hundreds to thousands), since it performs significantly faster than CVRS in this case. CVRS is used for problems in which a large number of restrictions have to be taken into account, which reflects many practical cases. Our research focuses on the optimizer CVRS.

To optimize the performance of CVRS for a certain customer, its behavior can be influenced by altering its configuration: a sequence of algorithms, including the parameters of each algorithm, used to solve the vehicle routing problem. The aim of this research is to develop a method that is able to systematically find good settings for a given customer. This research is a succession of the recent research by Van Dijk (2014) at ORTEC, who developed a tuning method for a loading algorithm. This raised the question at ORTEC whether similar techniques could also be applied to its routing algorithms. The technique by Van Dijk (2014) is not suitable in this case, due to the reasons we address in Section 1.2.

In this chapter we introduce our research. Section 1.1 provides the context of the optimizer CVRS. We discuss how this optimizer is working and how we can influence its behavior by changing the configuration. In Section 1.2, we introduce our problem and break it down in Section 1.3.

1.1 Context

CVRS is an optimization tool for optimizing transport planning. Given a set of tasks (transport orders) and a set of vehicles, CVRS aims at planning these tasks in those vehicles, while respecting the given restrictions (e.g., time windows) and optimizing the predefined criteria (e.g., driving distance). This problem is known as the Vehicle Routing Problem (VRP). CVRS' primary objective is to plan as many

[1] ORTECs technical platform common to all its software solutions: a scalable software architecture and application framework for real-time decision support systems

tasks of the given set as possible. Other objectives can be specified for specific customers, to meet their wishes. CVRS is designed to assist planners: people that have to solve practical VRPs. CVRS can assist them in increasing their performance by:

- Speeding up the planning process. CVRS can plan activities much faster than most (experienced) planners.
- Creating planning drafts for the planner to accept or to improve. The planner can use CVRS to create initial plans that could be modified to satisfy these planners' requirements that are hard to model.
- Handling complex planning activities (e.g., minimizing costs before minimizing distances, planning routes in areas where the planner is not familiar with, etc.). CVRS can optimize routes under various criteria and handle planning activities that are outside the capabilities of the planner.
- Optimizing existing routes. The planner can use CVRS to re-optimize existing routes. The sequence of planned tasks within the selected routes will be re-optimized by re-planning the tasks at hand.
- Generate alternatives (proposals) for the planner to choose from. Given a task and a set of routes, CVRS proposes routes in which the task can be planned.

In order to understand the settings that are relevant for CVRS, some background knowledge about the algorithms used in CVRS is required[2]. There are two main types of algorithms:

- Construction algorithms, which are used to create an initial solution.
- Improvement (or local search) algorithms, which are used to improve an existing solution.

To construct an initial solution, cheapest insertion is used. After this construction, improvement algorithms are applied to the initial solution. To influence the behavior of these algorithms, CVRS has to be configured. Before solving an instance, a configuration has to be specified. This configuration specifies a sequence of algorithms, including the parameters of each algorithm, used to solve the vehicle routing problem. A certain configuration is likely to perform different on different problem types. Therefore, configurations are adapted to make them suitable for the problems of a specific customer (i.e., a sequence of algorithms and parameter settings that are likely to solve the customer's problems well). Each of these algorithms has parameters to control the behavior of the specific algorithm. These parameters can also be varied to make the configuration even more suitable for a certain customer. There are three main parts, called levels, of the configuration that are highly tunable:

Level 0: The parameters of the construction algorithm For the construction algorithm, cheapest insertion, different parameters exist to influence the construction phase. Which parameter values to use in which case can be specified in the configuration, and could therefore be tuned. The search space consists of the possible values for the parameters.

Level 1: The sequence of improvement algorithms Within a configuration, we find a sequence of algorithms used to improve the solution. This sequence can be adapted to align it with the customer's situation. This implies that a sequence is chosen that is likely to solve the customer's problems well. Different algorithms can be used to improve the solution, e.g., 2-opt, merge and swap. Which algorithm(s) to use can be specified in the configuration, and could therefore be tuned. The search space consists of the possible sequences of improvement algorithms.

[2] Chapter 3 contains a detailed explanation of CVRS

```xml
<Template Strategy="Construction">
    <MaximumBatchSize value="2" />
    <SortingCriteria>
      <Criterion scale="0" direction="decreasing" name="AngleClosestRouteToTransport" />
    </SortingCriteria>
    <CheapestInsertion />
</Template>
<Template Strategy="Improvement">
    <2Opt EstimateWith="Distance" MinimumEstimatedGain="0" MaxNofIterations="100"
        OnlyAllowChangesWithinTheSameRoute="true"/>
    <LargeNeighborhoodMoveAndSwap OnlyModifyRoutesChangedInLastIteration="false"
        MinimumEstimatedGain="0" MaxNofIterations="200"
        MaxTravelTimeBetweenConsecutiveTasksInAGroup="300"
        OnlyAllowChangesWithinTheSameRoute="false" EstimateWith="Distance" />
    <2Opt EstimateWith="Distance" MinimumEstimatedGain="0" MaxNofIterations="100"
        OnlyAllowChangesWithinTheSameRoute="true"/>
    <LargeNeighborhoodMoveAndSwap OnlyModifyRoutesChangedInLastIteration="false"
        MinimumEstimatedGain="0" MaxNofIterations="200"
        MaxTravelTimeBetweenConsecutiveTasksInAGroup="300"
        OnlyAllowChangesWithinTheSameRoute="false" EstimateWith="Distance" />
    <2Opt EstimateWith="Distance" MinimumEstimatedGain="0" MaxNofIterations="100"
        OnlyAllowChangesWithinTheSameRoute="true"/>
</Template>
```

Figure 1.1: Simplified XML representation of the default configuration.

Level 2: The parameters of improvement algorithms A lot of algorithm-specific parameters are used to control the behavior of the improvement algorithms used. These parameters influence the trade-off between solution quality and computation time. Which parameter values to use in which case can be specified in the configuration, and could therefore be tuned as well. The search space consists of the possible values for the parameters.

The implementation of the configurations is done using so-called command templates. Command templates are XML files containing information about all three levels. The tuning of these configurations is currently done manually by the implementation consultants. Figure 1.1 shows an example of the current default command template.

It is important to note that the optimization tool is deterministic. That is, given a command template and a problem instance, it always produces the same solution. It has therefore no use to repeatedly solve a problem instance with the same command template.

1.2 Problem identification

Section 1.1 addressed the tuning options for the configuration in CVRS. Each customer has their own VRP problems and these problems could differ strongly between customers. We therefore distinguish different types of customers, grouped by the type of their problem. The optimal configuration is likely to differ among the problem types. For the same reason, a configuration performs differently for different customers. That is why configuration tuning is necessary and, at least, type-specific. From experience, ORTEC knows that the configuration has a large impact on the solution, both its quality and the required computation time. The need for good configurations is therefore present. However, finding good configurations is difficult, since it is unclear how a configuration performs for a problem, without executing (measuring) it. Moreover, measuring the performance of a single configuration on one problem instance of the customer takes roughly 5 to 15 minutes, which makes finding good configurations also very time consuming.

The time needed for a measurement, i.e., measuring the performance of a configuration, is the most important difference between our research and the research of Van Dijk (2014): minutes and seconds respectively. Since the tuning method developed by Van Dijk (2014) uses a lot of measurements, it is

not applicable in our case, due to the tuning time that would be needed.

Tuning the configuration, that is, configuring the routing algorithm, is currently done by the implementation consultants of ORTEC. They manually make tailor-made configurations: configurations adapted to problems faced by a certain customer. However, this is an undesired situation as it is a very time (and hence money) consuming task, caused by the facts that the environments differ strongly between the customers and that configurations are customer-specific. Next to this, there is no empirical evidence for the performance of the current configurations; they are largely based on the experience of the consultant. Therefore, ORTEC wants to automate the process of tuning these configurations.

It is important to keep in mind that, since ORTEC operates on many domains, an approach performing well on many domains, is considered as very valuable. Showing that a tuning method performs well on different domains significantly increases the value of this research for ORTEC.

1.3 Research problem

In this section, we further analyze and break down the problem identified in Section 1.2. First, in Section 1.3.1, we set the boundaries for our research. Next, in Section 1.3.2, we define our objective, and in Section 1.3.3 we clearly state our problem. We end this section with the research questions that guide our research.

1.3.1 Scope

The tuning method we want to develop has to operate within a certain context, whose boundaries we describe in this section.

To tune the configurations, we use the available improvement methods and parameters. We do not seek to develop new improvement methods, but limit ourselves to the currently implemented, and therefore useable, methods. Besides, we do not want to introduce new parameters, but stick to the currently available set of parameters. Summarized, we want to take full advantage of the current framework (CVRS) by deploying it with the best possible configuration for a certain customer.

We consider the time required to solve a problem instance as expensive, since it is the time a planner has to wait. We denote this time as the computation time. Currently, the configurations are tuned offline: not while solving a problem instance, but beforehand at moments when an abundance of computational resources is available. Therefore, tuning does not add to the computation time. We use a set of training instances that are similar to future, unseen instances. This is needed since the configuration is an input for CVRS, just as an instance of the problem. Another option might be to tune the configuration while solving the problem, known as online tuning. In this case, tuning does add to the computation time. As a result, solving the problem with online tuning requires more computation time compared to solving the problem with offline tuning. Combined with the desire to keep the computation time for the customer at a minimum, offline tuning seems to be more appropriate. This is also supported by the design of CVRS, because of its need for a pre-defined configuration. If online tuning would have been suitable for the problem instances solved with CVRS, a pre-defined configuration would not have been necessary. However, we review methods for both variants, since it is likely that we can adapt, or use logic incorporated in online tuning methods to make them suitable for offline tuning.

A first exploration of the currently used command templates does not give a strong indication about which level (see Section 1.1) of the configuration is most suitable for tuning. Since level 0 influences the way the initial solution is created, we expect a significant impact of the parameters on this level. In addition, our intuition says that changing the order of the sequence of the algorithms, level 1, has more influence on the outcome than tuning its parameters, level 2. Since we do not have clear empirical

evidence for this thought, we do not focus on a level right now. On the contrary, we try to develop a method that is able to address the tuning of all three levels. We notice some common patterns in the command templates, which indicates that certain combinations perform well. We take this into account when searching for and developing a new tuning method.

For our research, we have access to a single VRP instance of one of ORTEC's customers with 224 tasks. These tasks are spread over Western Europe and 23 vehicles are available for transporting them. From now on we refer to this VRP instance as $instance_A$. In addition, we have a set of 5 VRP instances of another customer of ORTEC. Each instance represents one day and contains about 1000 tasks. Each day, 10 vehicles are available. From now on we refer to these VRP instances as $instances_B$. Using these instances we are able to perform experiments and validate our tuning method. Due to the time restrictions of this research, we were not able to perform experiments with more (sets of) instances. Unfortunately, it is not possible to use benchmark instances with the software, such as the well-known Solomon (Solomon, 1987) instances.

1.3.2 Objective

The main objective of this research is to design and implement a method, able to automatically configure the routing algorithms to optimize their behavior on future instances. By this, the time-consuming manual tuning of the configuration will be reduced to a minimum. The quality of the solution should at least be comparable with the current quality. The computation time depends on the chosen configuration, since it specifies when to stop improving the solution (see Chapter 4). However, this is not based on time but on the state of the solution. The computation time with the tuned configuration should not exceed 15 minutes, since planners do not want to wait any longer (unless it significantly improves the quality of the solution). Hereby we discard the computation time needed to create a good configuration using our tuning method, since this is only done periodically, at moments with an abundance of computational resources. However, for practical reasons we do not want to spend more than a weekend (i.e., about 60 hours) on tuning. If we do not meet the given requirements, customers are likely to prefer the current configurations.

1.3.3 Problem statement

The optimization tool CVRS needs a method to automatically configure the routing algorithms to optimize their behavior on future, unseen instances. This, so-called, algorithm configuration problem can be defined as follows: "given a parameterized algorithm A, a set of problem instances I, and a cost metric c, find parameter settings of A that minimize c on I" (Hutter et al., 2011).

1.3.4 Research questions

We formulate a number of research questions to achieve our research objective in a structured manner. First, in Chapter 2, we introduce the VRP problem by exploring the comprehensive collection of solution methods for VRPs by means of a literature study. Maybe some of these methods would fit within our framework. However, most of the methods are not likely to be able to tune the configuration, since this would require a structural change to our framework (e.g., adding improvement methods or parameters). Afterwards, we therefore review methods that are able to tune configurations.

1. What literature is available related to vehicle routing problems and tuning configurations?
   (a) What solution methods are available for VRPs?
   (b) What configuration/parameter tuning methods are available?

Second, in Chapter 3, we want to analyze how the optimizer CVRS solves VRPs by using it and reading the manual. This helps us in understanding the impact of different configurations.

2. How does the optimizer CVRS solve VRPs?

Third, in Chapter 4, we want to analyze the configuration possibilities of CVRS. We discuss to what extent we can tune the configurations. In Section 1.2 we introduced the issue related to different types of problems. Our aim is the find a priori knowledge (e.g., algorithms that are suitable for a certain type of problem) that we can incorporate in our new tuning method. Here we use the experience of ORTECs experts. We also explore our performance landscape and look at the effect of various parameters.

3. What are the characteristics of our search space?
   (a) To what extent can we tune the configurations?
   (b) What useful information can we find in tailor-made configurations?
   (c) How does our performance landscape look like?

Fourth, in Chapter 5, we want to develop a tuning method that tunes the configuration by combining the knowledge obtained during our literature study as well as our exploration of the configuration.

4. What tuning method can we develop to automatically tune the routing algorithms?

Fifth, in Chapter 6, we want to test the performance of our new tuning method in various ways: (i) by comparing our method for tuning routing algorithms with alternative tuning methods, (ii) by comparing the tuned configurations with the configurations currently used, and (iii) by testing how the method performs on other problem domains (since ORTEC values general approaches, as mentioned in Section 1.2). To make this possible, we integrate the tuning method with CVRS.

5. How well does our new method perform?
   (a) How well does our new tuning method perform compared with alternative tuning methods?
   (b) How well do the configurations created with our new tuning method perform compared to other configurations?
   (c) How well does our new tuning method perform for other ORTEC products?

Finally, in Chapter 7, we want to present the conclusions and recommendations. We want to discuss how ORTEC can use our new tuning method in practice, such that its customers benefit from it. By discussing with ORTECs experts we find the best manner.

6. How can ORTEC implement our new tuning method?

The remainder of this thesis is structured such that the research questions are answered in sequence.

Chapter 2

Literature review

This chapter gives an overview of the literature about vehicle routing problems and configuration tuning. We start with a overview of methods that are used to solve vehicle routing problems in Section 2.1. Next, in Section 2.2, we address techniques that are used in the field of configuration tuning.

2.1 Vehicle routing problems

Over 50 years ago, Dantzig and Ramser (1959) introduced The Vehicle Routing Problem (VRP) as *The Truck Dispatching Problem.* It is a combinatorial optimization problem in which a number of customers has to be serviced with a fleet of vehicles given a set of constraints, while minimizing the total route cost. Many companies face this problem on a daily basis, for example if the supplying of supermarkets has to be planned.

Each VRP instance typically consists of many constraints, such as time windows, precedence relations, and vehicle capacity. Therefore, many variants of the VRP exist. However, a lot of research has concentrated on the classical VRP, a basic variant of the problem which can be adapted to meet real-life situations by adding restrictions.

The most common definition of the classical VRP is as follows. Let $G = (V,A)$ be an undirected graph where $V = \{0,1,\ldots,n\}$ is the vertex set and $A = \{(i,j) : i,j \in V, i \neq j\}$ is the arc set. Vertex 0 represents the depot, where m vehicles of capacity Q are located. The other vertices represent customers. Each customer $i \in V \setminus \{0\}$ has a non-negative demand $q_i \leq Q$. A cost matrix c_{ij} is defined on A. The problem consists of determining a set of at most m vehicle routes such that (i) each route starts and ends at the depot, (ii) each customer is visited exactly once by exactly one vehicle, (iii) the total demand of each route does not exceed Q, and (iv) the total routing cost is minimized.

The VRP is a generalization of the Travelling Salesman Problem (TSP). In a TSP, the goal is to find the shortest route from an origin, to each of the cities of the problem, and back to the origin.

We start with a discussion of the different variants of the VRP in Section 2.1.1. In Section 2.1.2 we explore the wide field of solution methods for VRPs.

2.1.1 Variants

The classical VRP does, in many cases, not reflect the real-life situation as a lot of constraints are missing. Capacity constraints were already included in the classical VRP but other common constraints include (Laporte, 1992):

- Time windows: location i must be visited within the time interval $[a_i, b_i]$ and waiting is allowed at location i (e.g., opening hours of a location). This variant is known as the Vehicle Routing Problem

with Time Windows (VRPTW).

- Precedence relations between pairs of locations: location i may have to be visited before location j. An example is the pickup of an order at location i which should be delivered at location j by the same vehicle, known as the Vehicle Routing Problem with Pickup and Delivery (VRPPD).
- Driving time restrictions: due to drivers' legislation, drivers have to comply with rules on driving time, breaks, and rests.

A dynamic VRP (DVRP), which may include any of the above constraints, is a VRP that can change while solving the problem. Traffic jams or unexpected arrivals as well as cancellations of orders are examples of such changes.

A typical, practical VRP instance contains several constraints of different types (e.g., time windows and precedence relations). Since the aforementioned abbreviations are insufficient in those cases, problems with many types of practical constraints are called rich VRPs. Most practical instances are therefore rich VRPs.

2.1.2 Solving VRPs

The VRP is a NP-hard problem for which it is hard to determine sharp lower bounds on the objective value (Cordeau, Gendreau, Laporte, Potvin, & Semet, 2002). Exact algorithms using implicit enumeration will therefore converge slowly, which makes them incapable of solving realistic problem sizes with a constant success rate in an acceptable time. The best known exact algorithms are able to handle approximately a hundred vertices (Baldacci, Christofides, & Mingozzi, 2008), but real instances often exceed this size. As a consequence, research has focused on heuristics (Laporte, 2007). Heuristics are also easier to adapt (e.g., adding restrictions), which is needed in order to meet real-life situations. The dynamic programming heuristic by Kok, Meyer, Kopfer, and Schutten (2010), for example, takes the full European social legislation on drivers' driving and working hours into account. Nevertheless, solution methods for rich VRPs are scarce.

To structure the remainder of this section, we use the classification of solution methods for the VRP by Laporte (2007) as a guideline. First, we review conventional methods, divided into exact algorithms and classical heuristics. With the term "classical" we refer to heuristics that do not allow the objective function to deteriorate in a consecutive iteration. Heuristics that do allow this, are described later in this chapter, and we call them metaheuristics. Both classical heuristics and metaheuristics search within a search space of problem solutions. The so-called hyper-heuristics, on the other hand, search within a search space that contains heuristics. For this reason, we address them in Section 2.2, where we discuss configuration tuning methods.

Exact algorithms

Exact algorithms exist to solve the VRP, but their success in solving realistic instances with larger size is limited. Direct tree search methods, dynamic programming, and inter linear programming are the three main types of methods within this category.

Direct tree search Direct tree search methods solve the VRP by sequentially building routes by using a branch and bound tree. Christofides and Eilon (1969) developed one of the first algorithms using a direct tree search method. Their algorithm branches on arcs, and branches are created by either including or excluding an arc in the solution. Later, Christofides (1976) developed a branch and bound algorithm that branched on routes instead of arcs. This resulted in wide search trees with a limited depth. Only

easy or small instances could be solved with these algorithms. The introduction of two methods that were able to derive sharp lower bounds (Christofides, Mingozzi, & Toth, 1981) considerably enhanced the performance of these algorithms. Fisher (1994) used this knowledge as well, which resulted in a method able to solve an instance with 71 customers.

Dynamic programming Dynamic programming (DP) is an optimization approach that can solve complex problems by dividing them into a sequence of simpler sub-problems. These sub-problems are solved in multiple stages so that in each stage a part is added to the partial solution. In the last stage, the optimal solution is found. The following DP algorithm is based on the DP formulation for the Traveling Salesman Problem by Held and Karp (1961). Let V be a set of nodes representing all customer locations with 0 being the departing node (e.g., the depot). Given $S \subseteq V \setminus \{0\}$ and $i \in S$, let $C\{S,i\}$ be the cost of the best route that starts in node 0, visits all nodes in set S and ends in node i. Then the recurrence relation is $C(S,i) = c_{0i}$ for $S = \{i\}$ and $C(S,i) = min_{j \in S \setminus \{i\}}[C(S \setminus \{i\},j) + c_{ji}]$ for S otherwise. The second part of the recurrence relation can be explained as follows. Consider the shortest path from node 0 to node i, visiting all nodes from $S \setminus \{i\}$ which has node $j \in S$ immediately preceding node i. Since the nodes in $S \setminus \{i\}$ are visited in optimal order, the length of this path is $C(S \setminus \{i\},j) + c_{ji}$. By taking the minimum over all choices of j we obtain the minimum of $C(S,i)$. The ultimate goal is to find the minimum cost of a complete tour, terminating in node 0, which is given by $C(V,0) = min_{i \in V \setminus \{0\}}[C(V \setminus \{0\},i) + c_{i0}]$. In each stage, the partial solutions are extended with an extra node. The number of stages is the number of nodes that are available in DP. Each stage can have multiple states, which are partial solutions of the main problem. The discussed DP algorithm creates only one route and applying it to the VRP is therefore not straightforward. Gromicho, van Hoorn, Kok, and Schutten (2012) use the giant-tour representation (GTR) of vehicle routing solutions to apply the approach to the VRP. In a GTR all routes are connected such that they form one giant route. By this, the VRP is transformed into a sequencing problem. This makes it possible to use the DP formulation of the TSP to solve it. The framework of Gromicho et al. (2012) ensures that the DP solution of the GTR is feasible for the original VRP.

Integer linear programming Most research on exact algorithms for the VRP has been done in the field of integer linear programming (Laporte & Nobert, 1987), which has resulted in variety of methods in this category. Vehicle flow formulations are by far the most widely used among the integer linear programming (ILP) methods. Variables indicating how many times edge $[i,j]$ appears in the solution are used in the two-index variant by Laporte, Nobert, and Desrochers (1985). The three-index variant adds the vehicle making the route to this variable (x_{ijk}) (Fisher & Jaikumar, 1978).

Set partitioning formulations, first suggested by Balinski and Quandt (1964), are rarely used in practice because of their large number of variables. However, the most successful VRP algorithms partially use a set partitioning formulation (Laporte, 2009). Examples of such algorithms are found in Fukasawa et al. (2006) and Baldacci et al. (2008).

Commodity flow formulations, such as the two-index two-commodity flow formulation for the capacitated VRP by Baldacci, Hadjiconstantinou, and Mingozzi (2004), use additional variables for both the vehicle load on an edge and the remaining capacity of the vehicle travelling on an edge. The authors show that this formulation allows to solve instances up to 100 customers, although with mixed success.

Classical heuristics

Classical heuristics can be subdivided into two categories: construction heuristics and improvement heuristics. The main difference between these two is that construction heuristics do not start with an

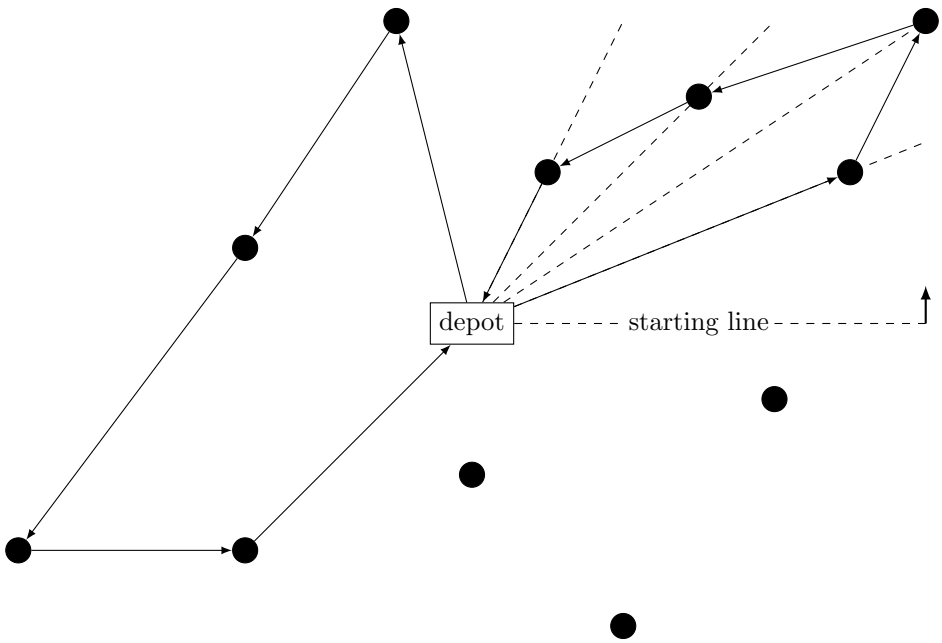

Figure 2.1: An example of the sweep algorithm, in which the capacity of a vehicle is 4 customers and the sweep direction is counter-clockwise.

initial, feasible solution whereas improvement heuristics do start with a feasible solution which they try to improve.

Construction The most well-known construction heuristic to solve a VRP is probably the saving algorithm by Clarke and Wright (1964). The main idea is to merge routes based on the generated savings by this merge. Initial routes are constructed from the depot to the customer and directly back to the depot (so one customer per route). Next, the algorithm calculates the possible savings $s_{ij} = c_{i0} + c_{0j} - c_{ij}$ by removing the arcs $(i,0)$ and $(0,j)$ and adding the arc (i,j), if this results in a positive saving and a feasible merged route. The merge yielding the largest saving is implemented. This process iterates until profitable and feasible savings are not possible anymore. The simple implementation of this heuristic and the ease with which additional restrictions can be added explains its popularity. Several improvements to the saving algorithm have been proposed. Gaskell (1967) and Yellow (1970) try to create more compact routes by adding a positive parameter λ to savings formulation: $s_{ij} = c_{i0} + c_{0j} - \lambda c_{ij}$. The enhancement of Altinel and Öncan (2005) uses a modified savings function that includes the customer demands impact.

Another well-known heuristic is the sweep algorithm by Gillett and Miller (1974). Starting with a line rooted at the depot, it sweeps this line either clockwise or counter-clockwise. Customers that are encountered while sweeping this line are added to the route. A new route starts once it is infeasible to include the next customer to the current route, for example because of capacity restrictions of the vehicle. Figure 2.1 visualizes the algorithm. This method makes sure that only non-intersecting routes are generated. This method is a so-called cluster first, route second method. These methods are characterized by the fact that they first form groups of locations, clusters, and try to build a route based on these clusters afterwards.

Fisher and Jaikumar (1981) presents an algorithm that consists of two phases. First, seed points (i.e., points that will end up in disjoint routes) are chosen and customers are divided in clusters by solving a generalized assignment problem (GAP). How to determine these seed points is not made clear in the original article, but some articles address this issue (Bramel & Simchi-Levi, 1995 and Baker & Sheasby, 1999). Second, a TSP is solved on each cluster.

CVRS uses cheapest insertion, based on the sequential insertion heuristic by Solomon (1987), as its construction heuristic. Routes are initialized with a seed location, selected by, for example, the location farthest away from the depot. The remaining locations are added at the cheapest insertion point in the route constructed so far, until the route is full (e.g., capacity constraints). The insertion cost is defined as $c_{ik} + c_{kj} - c_{ij}$ for an insertion of location k. This process repeats until all locations are serviced.

Improvement Improvement heuristics take a feasible route as a starting point. Then, they try to improve the route by either (i) intra-route or (ii) inter-route changes. Intra-route heuristics switch the position of one or more customers within a route; inter-route heuristics move one or more customers between different routes.

The 2-opt heuristic by Croes (1958) is an intra-route improvement method that generates a 2-optimal route: a route that cannot be shortened by exchanging two arcs. This is achieved by removing crossings (arcs that intersect) from routes, as crossings are never optimal in the classical VRP with symmetric cost matrix. However, because of, for example, time-windows, the optimal route could contain crossings in other VRP variants. A related method is Or-opt (Or, 1976), where a number of consecutive vertices are relocated while maintaining the orientation of the original route.

Thompson and Psaraftis (1993) describe a b-cyclic, k-transfer scheme in which b routes are selected for a circular permutation and for each route k customers are moved to the next route of the permutation. Computational experience shows that combinations of $b = 2$ or $b = variable$ and $k = 1$ or 2 seem to produce good results. Slightly modified approaches of this idea are used in the improvement phase of CVRS.

Numerous other inter-route improvements heuristics have been developed, including ejection chains (Glover, 1992), GENI (Gendreau, Hertz, & Laporte, 1992), λ-interchange (Osman, 1993), and CROSS (Taillard, Badeau, Gendreau, Guertin, & Potvin, 1997).

Metaheuristics

The exploration of the solution space is much more thorough within metaheuristics compared to classical heuristics. Within metaheuristic, moves that are inferior or infeasible can also be accepted in order to escape from local minima. Three main categories exist within this field: (i) local search, (ii) population search, and (3) neural networks.

Local search Local search methods are methods that explores the neighborhood of the current solution and moves each iteration to a solution in that neighborhood. Different types of methods exist within this category to define and explore neighborhoods.

Tabu search methods remember properties of previously visited solutions and make sure that those are not considered again in subsequent iterations. A metaheuristic based on this approach is TABUROUTE, developed by Gendreau, Hertz, and Laporte (1994). This algorithm uses neighborhoods that are found by "repeatedly removing a vertex from its current route and reinserting it into another route". They also allow intermediate infeasible solutions to escape from local minima, which is not common for tabu search methods. Taillard (1993) presents an approach that decomposes a VRP into independently solvable subproblems. This speeds up the applied iterative search method, in their case tabu search. The strengths of the unified tabu search algorithm (UTSA) by Cordeau, Laporte, and Mercier (2001) lie in its flexibility and robustness. Using a simple exchange procedure, controlled by tabu search, neighborhoods are explored and good quality solutions are found. Toth and Vigo (2003) propose a granularity concept. Long edges (i.e., edges with a length exceeding a certain threshold) are never considered by the search process as it is not very likely that they belong to the optimal solution.

Another branch of local search metaheuristics is based on variable neighborhood search. Here, the algorithms try to find local optima, and switch to another neighborhood once a local optimum is reached. The two-phase variable neighborhood search heuristic by Kytöjoki, Nuortio, Bräysy, and Gendreau (2007) uses a variable neighborhood search scheme to guide the improvement heuristics. Other applications of variable neighborhood search to the VRP are the two-stage hybrid algorithm by Bent and Van Hentenryck (2004) and a very-large scale neighborhood search algorithm by Ergun, Orlin, and Steele-Feldman (2006). A well-known heuristic based on variable neighborhood search is the heuristic by Pisinger and Ropke (2007). We discuss this heuristic in Section 2.2.1, since it is strongly related to configuration tuning.

Population search Population search methods, such as genetic algorithms (Holland, 1975), simulate the process of natural selection. By mutating the properties of candidate solutions from a population of solutions, the population evolves towards a new generation consisting of hopefully better solutions. Baker and Ayechew (2003) applied this method to the classical VRP problem. Their results show that the method can be competitive with other heuristics, both in computation time and solution quality. Often genetic algorithms are combined with local search, as described by Rochat and Taillard (1995), Mester and Bräysy (2007), and Vidal, Crainic, Gendreau, and Prins (2014).

Ant colony optimization has received a lot of attention for VRPs. These methods are based on the behavior of ants. Edges that appear frequently in good solutions are remembered as good edges and they are more likely to be included in other solutions. Early attempts based on these ideas applied the method to the classical VRP (Bullnheimer, Hartl, & Strauss, 1999), and their promising results led to ant colony optimization methods for variants of the classical VRP (Gambardella, Taillard, & Agazzi, 1999).

Many other population search methods exist, such as genetic programming or evolution strategies, but they are not noteworthy in the light of the VRP, since, to our knowledge, they have not been applied successfully.

Neural networks Neural networks mimic the way biological neural systems, such as the brain, operate. This concept has only received little attention in the context of vehicle routing. Ghaziri (2004) and Schumann and Retzko (1995) have both applied self-organizing maps, a type of neural network, but they have not been able to find good solutions in an efficient way. The same holds for the approach of Potvin and Robillard (1995). To our knowledge, this line of research has received little attention in recent years. Combined with the unsuccessful early attempts, it is plausible that the opportunities of neural networks for the VRP are limited.

2.1.3 Conclusion

Exact algorithms are not applicable for our research as none of the described methods is able to solve instances of a realistic size, especially not with our limitations on computation time. As a result, we focus on heuristic approaches, which are currently used by CVRS.

The classical heuristics require low computational effort, but they are incorporating many restrictions is complicated. This makes them unappealing for practical situations in which numerous side constraints exist.

Metaheuristics are more flexible, which makes them more useful in practice, but this is done at the expense of computation time. None of the metaheuristics described in our review are solely based on simple heuristics as we have in our framework. Instead, they are mostly tailor-made for a specific VRP variant, built completely from scratch. Using them within our framework is therefore not straightforward.

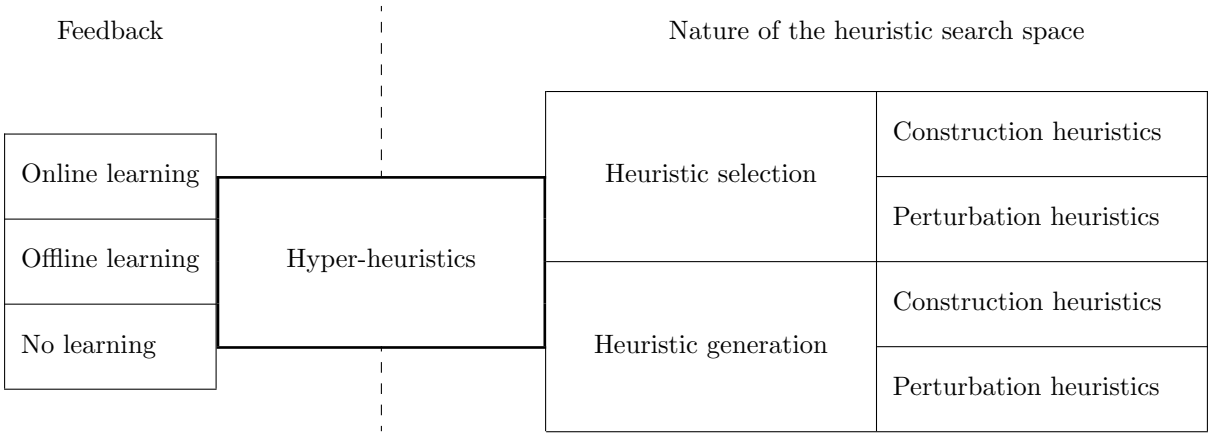

Figure 2.2: A classification of hyper-heuristic approaches (Burke et al., 2010).

Summarized, none of the discussed methods strongly resembles our complete framework, making them unsuitable to solve our problem.

2.2 Configuration tuning

As mentioned before, we have three levels in the configuration to tune. First, we discuss methods to tune the sequence of our improvement algorithms, which we called level 1. Therefore we need methods that use a search space of heuristics rather than solutions (as metaheuristics do). Applicable literature can be found in the field of hyper-heuristics, which we discuss in Section 2.2.1. Second, we discuss general tuning methods, which can be applied to all levels, in Section 2.2.2.

2.2.1 Hyper-heuristics

A hyper-heuristic is defined as "an automated methodology for selecting or generating heuristics to solve hard computational search problems" (Burke et al., 2010). This research field arose from the need to raise the level of generality at which search methodologies can operate. This term was first used in the context of combinatorial optimization to define "heuristics that choose heuristics" (Cowling, Kendall, & Soubeiga, 2001). In this context, hyper-heuristics operate on a search space of low-level heuristics. Low-level heuristics are generally widely known heuristics that are able to solve (parts of) VRPs, such as those discussed in Section 2.1.2. From this, given a particular problem instance, they select and apply suitable heuristics. This seems to fit with our need for a method that selects appropriate low-level heuristics, given the problem instance at hand.

Burke et al. (2010) propose a classification for hyper-heuristics, which we will use to structure our discussion on hyper-heuristics. Their classification uses two dimensions: (i) the nature of the heuristic search space, and (ii) the source of feedback during learning.

From the definition of a hyper-heuristic, we conclude that there are two categories of hyper-heuristics: heuristic selection and heuristic generation. Figure 2.2 shows that both categories are also present in the proposed classification. We therefore consider these two categories as the most fundamental, and the remainder of this section is structured along these.

Figure 2.2 provides us with more information about the different types of hyper-heuristics. Within both heuristic selection, and heuristic generation, we find another categorization. This categorization concerns the nature of the search space (i.e., the type of low-level heuristics), namely between construction and perturbation heuristics. Perturbation heuristics are similar to the improvement heuristics (see

Section 2.1.2). Methods based on construction heuristics build their solution incrementally: an empty solution gradually evolves to a complete solution by intelligently selecting and applying construction heuristics from a pre-defined set. On the other hand, methods based on perturbation heuristics do not start with an empty solution. They aim to improve a candidate solution by selecting and applying perturbation heuristics in a clever way. Hyper-heuristics may also consider both construction and perturbation heuristics. We label these as hybrid methods.

The other dimension, the source of feedback, concerns the learning mechanism. This describes the way feedback from the search process is used. In this classification, we find three groups: (i) online learning, (ii) offline learning, and (iii) no learning. The first group comprises methods that learn while solving an instance of a problem. Here feedback is received during the search process of a specific instance. Therefore, instance-specific properties can be taken into account, resulting in a selection of low-level heuristics that is tailor-made for the problem instance. On the other hand, offline learning approaches use a set of training instances. These training instances should be representative for other unseen problem instances of the same type. The idea is to gather knowledge for a generalized problem of a particular type to develop a heuristic that, hopefully, performs well on any instance of the same type. Obviously, the latter group, no learning, does not use any feedback. In this case, choices are made based on predetermined rules that do not depend on knowledge gathered in previous iterations (e.g., randomly).

From the previous paragraph, we conclude that using hyper-heuristics does not, implicitly, make a choice between online or offline tuning either: a hyper-heuristic could use either method. In our case, we tune offline, as stated in Section 1.3.1. However, we might be able to use some logic incorporated in online tuning methods. Here it is important the keep a structural difference between online and offline tuning methods in mind. As introduced in Section 1.3.1, the time an online tuning method needs is part of the computation time for a problem instance. As a consequence, online tuning methods typically focus on parts of the search space that are likely to perform well. By this, the risk that time is spent on measuring bad performing configurations is reduced. For offline tuning methods, the tuning time does not add to the computation time. Offline tuning methods can afford to measure configurations in parts of the search space where the performance is still unknown, to reveal the performance landscape in these areas. Since we opt for offline tuning, we have the opportunity to explore unknown parts of the search space. If we (partly) use online methods, we must ensure that we do not focus on known parts of the search space only.

We continue this chapter with a discussion of examples (in the application domain of vehicle routing, if possible) of heuristic selection methods and heuristic generation methods. For an exhaustive survey, we refer to Burke et al. (2013).

Heuristic selection methods

Heuristic selection methods solve a problem by choosing promising combinations of existing low-level heuristics, both construction and perturbation heuristics.

Among the heuristic selection methods (partly) based on construction heuristics, we find a hill-climbing based hyper-heuristic by Garrido and Castro (2009). Hill-climbing is a local search method in which incremental changes to an initial solution are made until no further improvements can be found. A collaborative framework chooses a combination of simple low-level heuristics, based on the nature of the problem at hand (online learning). It is a hybrid method since it considers both construction and perturbation heuristics. Tests of the method on the CVRP show that this method delivers stable and good quality solutions for various types of problems. This implies that the framework is able to adapt itself to the problem instances and chooses suitable combinations of low-level heuristics. In a follow-up paper, an

evolutionary hyper-heuristic is proposed (Garrido, Castro, & Monfroy, 2009). This hyper-heuristic is very similar, also in terms of performance on the CVRP, to the previously mentioned hill-climbing based hyper-heuristic. Garrido and Riff (2010) use the idea of an evolutionary hyper-heuristic to solve the DRVP. To be able to take the unexpected changes that might occur in a DVRP into account, they introduce a third type of low-level heuristic: noise heuristics. Again it achieves competitive results. Finally, HyperPOEMS (Mlejnek & Kubalík, 2013) is an improved evolutionary hyper-heuristic. An evolutionary-based iterative search algorithm controls the process of autonomously searching suitable combinations of low-level heuristics for a particular problem instance. Results indicate that it outperforms the previously mentioned hyper-heuristics for the CVRP. Although the aforementioned hyper-heuristics are, at least partly, based on construction heuristics, the main idea could still be used in our case as the effect of the order of the sequence is taken into account here.

Pisinger and Ropke (2007) present a method that is based on perturbation heuristics. In their adaptive large neighborhood search (ALNS) "a number of simple algorithms compete to modify the current solution". In each iteration a perturbation heuristic, consisting of one algorithm to destroy and one algorithm to repair the current solution, is chosen. An adaptive layer that uses roulette wheel selection, makes this choice, based on the low-level heuristics performance during previous iterations, so it learns online. A local search framework (e.g., simulated annealing or tabu search) decides whether or not to accept the new solution, created by the chosen heuristics. The generality of the ALNS framework makes it possible to use the method for a wide spectrum of VRPs. The fact that the method was able to improve several well known solutions of benchmark instances, shows that it is promising. However, the framework does not take the sequence of the heuristics into account. Therefore the effect of the order of the sequence, which we consider as important, is neglected. The coalition-based metaheuristic (CBM) by Meignan, Koukam, and Créput (2010) is a self-adaptive and distributed metaheuristic. Several agents concurrently modify the solution by choosing a low-level heuristic. The selection of a heuristic is adapted dynamically using learning mechanisms, which are applied during the optimization. To assess the performance of CBM, experiments on the VRP have been conducted. This shows that the heuristic is competitive with powerful heuristics.

Heuristic generation methods

Heuristic generation methods tackle a problem by searching and selecting from a space of basic building-blocks of heuristics. These building-blocks are obtained by decomposing heuristics into their basic components. Heuristic generation methods return, next to the solution, a reusable heuristic that can be used to solve similar yet unseen problems. At first sight this seems to fit perfectly with our problem and the desired solution, as we can consider our improvement algorithms as components of our configuration (our complete improvement algorithms).

To our knowledge, only a few attempts have been made to apply a heuristic generation method to the VRP. These efforts take a grammatical evolution (GE) (O'Neill & Ryan, 2001) approach. GE is related to genetic programming: it is an evolutionary algorithm that can evolve complete programs. Drake, Kililis, and Özcan (2013) propose a GE hyper-heuristic that evolves components of the well-studied metaheuristic VNS. Sabar, Ayob, Kendall, and Qu (2013) present a framework that uses GE, "which takes several heuristic components as inputs and evolves templates of perturbation heuristics". Although both methods are categorized as heuristic generation methods, neither reuses the generated heuristic on other instances of the same type. Instead, the heuristic is applied for each instance individually, making it hard to predict if the generated heuristic is suitable for reuse.

In other application domains we find methods that generate heuristics, which are thereafter tested on reusability. Burke, Hyde, and Kendall (2012), for example, present a GE method for the bin packing

problem that produces good quality perturbation heuristics. Only one training instance is used in the offline learning process, which is used for the evolution of the heuristic. Still, each run on the training instance(s) generates a new, and possibly different, heuristic. This set of heuristics maintains its performance on new problem instances for the same type, i.e., they are appropriate for reuse. A method, to choose the most suitable heuristic for the problem type out of this set, is not provided. The difference in computation time for one iteration in the case of Burke et al. (2012) and our case differs strongly, less than a second and 5 to 15 minutes respectively. In the bin packing case, hours of tuning were needed to find a suitable solution. As it seems plausible that a comparable part of the search space should be measured in our case. As a consequence, it would take weeks to find a feasible solution in our case. Although we did not restrict the time for offline tuning, weeks is obviously too long for the scope of this research. Other methods are largely based on genetic programming as well, making them unsuitable as well given the issues with tuning time that will occur.

Conclusion

We discussed several methods to tune the sequence of the algorithms in the configuration. In general, the methods choose the most applicable low-level heuristic out of a set of low-level heuristics, based on the nature of the problem on hand. To achieve this, most methods use online learning. However, some methods seem to be able to make them applicable for offline tuning by slightly modifying them (as we already expected).

In general, the heuristic selection methods use online learning mechanisms. The use of a certain low-level heuristic differs between different problem types. We might therefore be able to apply the methods as offline tuning methods, and use the results for problem instances of the same type. Since these methods produce sequences of heuristics for a particular problem instance, we would likely end up with many different sequences. Therefore, we would have to solve the difficult problem of creating the most promising sequence out of a large set of sequences (evolved from all instances of the training set). We could simplify this problem by selecting, instead of creating, the most promising sequence. By enumerating all sequences for all instances, we could then select the sequence that performs best on average. However, it is far from certain that the sequence we select in this case, is a good sequence on average. That is, we might have found a completely different, better sequence if we aimed at creating a single sequence for all instances. This approach seems therefore unsuitable.

The use of a heuristic generation methods seems more appropriate. Several methods show that the generated heuristics are indeed suitable for reuse. However, none of the methods develops one heuristic per problem type. Instead, they provide a set of heuristics that, individually, perform well on a certain problem type. If we would use these methods in our case, we have to find the most suitable heuristic of this set. We end up with the same problem as for heuristic selection methods. In addition, these methods would require weeks of computation time, which does not suit this research.

We could adapt the before mentioned methods to align them more with our needs, for example by using methods that are able to choose the most promising heuristic out of the created set. Nevertheless, hyper-heuristics are generally used for problems with a much smaller search space. It is therefore likely that the computation time exceeds our limits. Moreover, tuning the settings of the algorithms (level 2) is not addressed with these methods, so we would need an additional method. As a consequence, we would have to combine these methods, which could be complicated as well.

2.2.2 Automated tuning methods

Different configurations typically result in different solutions, so tuning them has a high likelihood of influencing the performance. Literature on automated tuning techniques covers this subject. Since tuning

is time-consuming and hard to do by hand, automation is very likely to be beneficial.

Parameters can be divided into two categories: numerical and categorical. The distance between numerical parameter values can be used as a metric, but this is not possible for categorical parameter values. In the case of tuning numerical parameters, this metric is used to guide the search. Categorical parameters are more difficult to tune, as many samples are required to spot parameter superiority (Dobslaw, 2010). Assessing this distinction is therefore important and we will take it into account in our discussion.

Based on Hutter, Hoos, Leyton-Brown, and Stützle (2009), we define our problem as follows. Let $A(\theta,\Pi)$ be the algorithm we want to optimize, with its parameters $p_1,\ldots,p_k$ that returns the objective value for a minimization problem. Θ_i is the domain, i.e., the possible values, for each parameter p_i. The space of all possible configurations is denoted by $\Theta \subseteq \Theta_1 \times \cdots \times \Theta_k$. The quality of the configuration $\theta \in \Theta$ is $A(\theta,\Pi)$, which is a deterministic quality measure, which depends on the problem instance π on hand. We denote the training set by Π. We aim to find the optimal configuration $\theta^* \in \text{argmin}_{\theta\in\Theta} A(\theta,\Pi)$.

Unfortunately, no universally applicable tool to solve this problem exists, but recent developments have led to significantly improved methods (Dobslaw, 2010). In general, but especially in our case, tools should be easy to apply without the need for a priori knowledge about parameter interactions, as this can be difficult to determine. Since we did not yet find a suitable tuning method for level 1 in the field of hyper-heuristics, we again take this level into consideration. Our search space for the improvement phase has a hierarchical structure: some parameters are used to select an improvement algorithm, whereas others control the behavior of an improvement algorithm. Some parameters are conditional. For example, only if heuristic H is chosen, parameter p_i is used. Also, evaluating a configuration is expensive (see Section 1.2), indicating the need for a method that severely limits the number of configuration evaluations.

To structure this topic, we use the recent research of Dobslaw (2010) as a starting point. We divide the methods into two categories: (i) model-based and (ii) model-free. Model-based approaches run the algorithm several times with different parameter values. These experimental results are used to build a model of the relation between the parameter values and the performance. This model is used to decide which parameter values to use for subsequent measurements. Model-free approaches do not build such a model, but parameter values are chosen using heuristic rules.

Model-based

Model-based approaches use their evaluations to build a model, reflecting the performance landscape or response surface. Response surface methods (RSMs) are a good example here. These models make interpolation and extrapolation possible. Most methods use this characteristic to guide their search. Our solution spaces, and hence response surfaces, are likely to be irregular, mainly due to the presence of categorical parameters. Moreover, our solution is optimized on multiple objectives, which means that it is hard to express the quality of a solution in a single value. This makes the use of response surfaces complicated.

Efficient Global Optimization (EGO) (Jones, Schonlau, & Welch, 1998) is based on expected improvement. A response surface model is built by iteratively evaluating the performance of promising configurations. To find promising configurations, the expected improvement of the objective function is used. That is, instead of directly evaluating the performance, which would be too expensive, an expectation is calculated using a simpler, less-expensive, and deterministic model. However, the performance of a configuration is not necessary deterministic, since some improvement algorithms contain randomized parts. An extension to EGO, Sequential Parameter Optimization (SPO) (Bartz-Beielstein, Lasarczyk, & Preuss, 2005), overcomes this problem and aims at optimizing stochastic algorithms. It iteratively analyzes points in the search space, with the performance evaluated by a stochastic model. The points

with the highest expected improvement, taking the uncertainty into account, are selected for the next iteration. Due to the stochastic nature, promising points are evaluated several times, resulting in a computational resource intensive method. Other drawbacks are that the current version only allows numerical parameters, and that applying it to a set of training instances quickly becomes impractical. Another method is the knowledge gradient policy by Frazier, Powell, and Dayanik (2008). It myopically optimizes the value of information, by directing the search towards areas from which little information is known. Mes, Powell, and Frazier (2011) proposes a similar policy, which tries to learn about many alternatives from a single measurement by using shared characteristics. This requires a priori knowledge about the aggregation structure of the search space. Several other sequential measurement techniques are available (e.g., Mockus (1994)), which are all, somehow, based on response surfaces. Applying them in our context, with both categorical and conditional parameters, is therefore not straightforward.

The drawback of many of the discussed model-based methods is that single problem instances are optimized, which is omitted by F-race (Birattari, Stützle, Paquete, & Varrentrapp, 2002). F-race starts with the evaluation of a set of possible configurations, so discrete sets of parameters are needed. Subsequently, the idea of racing is applied. That is, poor performing configuration should receive low attention (i.e., computational resources), which is achieved by eliminating statistically poor performing configurations. The initial evaluation of many configurations is expensive and causes scalability problems. To overcome this, Balaprakash, Birattari, and Stützle (2007) present a modified racing algorithm. Although this allows for configurations with more parameters, the inclusion of categorical and conditional parameters is not fully supported. The principle of racing can also be used as an extension to other methods, as we discuss later in this section.

The discussed model-based methods entail two major problems: (i) they are only applicable to numerical parameters, and (ii) they evaluate their performance on one problem instance, instead of on a set of problem instances. However, some techniques can be used, as an addition to the discussed methods, to overcome these problems.

To tackle the first problem, we need to be able to use regression related techniques on categorical data. In this light, the k-nearest neighbor algorithm (Buttrey, 1998) is, among others, suitable. It estimates the performance of unseen points based on the performance of the k closest previously evaluated points. Hence, a distance metric between categorical variables is needed to determine the proximity. Methods as, for example, the Hamming distance (Hamming, 1950) and the ordered-based sequence similarity (Gómez-Alonso & Valls, 2008), have both been successful in this area. Another suitable approach is the use of random forests (Breiman, 2001). It is based on the use of regression trees: trees similar to decision trees, but with leaves that contain information about the performance instead of a classification. These trees are built using gathered measurement data and show the relation between parameters choices and performance. The performance of unseen configurations can therefore be estimated, a principle similar to the use of response surfaces.

An obvious method to solve the second problem would be to evaluate the performance on multiple (N) of instances, and report the overall performance (e.g., the mean or median). However, the choice of N is important yet difficult: too few training instances leads to poor generalization possibilities, i.e., poor performance on unseen instances, and too many training instances would make the tuning process unnecessarily long. We can adaptively choose N, based on the configuration on hand by evaluating poor configurations on only a few instances, but promising configurations on more instances (as is done by FocusedILS, a model-free approach we discuss later). A different method is presented in Leyton-Brown, Nudelman, and Shoham (2009). In addition to characteristics of the configuration, features of the problem instances are taken into account as well (e.g., the number of locations to visit). By this, problem instances are divided into categories, and this information is subsequently used in estimating the performance of a configuration on a problem instance. Combinations of configurations and problem

instances that do not look promising are not evaluated, which limits the number of evaluations.

To our knowledge, the only method combing a model-based method with some of the proposed additions is Sequential Model-Based Algorithm Configuration (SMAC) (Hutter et al., 2011). It uses random forests to cope with categorical parameters, and it evaluates the performance on a set of problem instances. At first sight, no issues withhold us from applying the ideas of this method in our case. Since it has been introduced recently, little evidence exists regarding its performance, but the authors' first experiments hold great promise.

Model-free

Since model-free approaches do not build a model of the performance landscape, extrapolation is virtually impossible and promising areas are hard to determine. As a result, evolutionary algorithms are often applied in this category, and randomness still plays an important role when selecting a new experiment.

A well-known and widely used method here is the covariance matrix adaptation evolution strategy (CMA-ES) by Hansen and Ostermeier (2001). However, it is a numerical optimization method, so the categorical and conditional parameters in our case make that we have to apply complicated modifications. We do not know how these modifications affect the performance of the method.

Estimation of distribution algorithms (EDAs) can be regarded as an enhancement of evolutionary algorithms. Instead of solely looking for good points in a search space, it estimates distributions of promising points (Eiben & Smit, 2012), limiting its use to continuous domains. Subsequently, this information is used to gain insight into the relevance and sensitivity of different parameters. This idea is exploited by Relevance Estimation and Value Calibration (REVAC) (Nannen & Eiben, 2006). Probability distributions are, implicitly, created for parameters, based on previous evaluations.

A method that has, on a high level, a lot in common with tuning by hand is Parameter Iterated Local Search (ParamILS) (Hutter, Hoos, Leyton-Brown, & Stützle, 2009). It starts with an initial configuration, usually a default configuration or a configuration based on experience. Next, the neighborhood, defined as the change of a single parameter value, is explored by conducting iterated local search. This iterated local search procedure searches the neighborhood randomly and accepts configurations once they yield an improvement on a set of problem instances. A random probability determines whether or not to re-initialize the search, to escape from local minima. Since a discrete set of values for all parameters is needed, categorical parameters can be included. ParamILS supports conditional parameters by excluding neighborhoods with changes in inactive parameters. BasicILS and FocusedILS are two variants of ParamILS. BasicILS uses a fixed number of problem instances for the evaluation of each configuration; FocusedILS adaptively chooses this number.

The last model-free approach we discuss here is the gender-based genetic algorithm (GGA) by Ansótegui, Sellmann, and Tierney (2009). The gender-based approach reduces the number of function evaluations by half. Moreover, this reduction does not result in a significant loss in solution quality. Relations between parameters have to be specified in advance, using so-called variable trees, in which both numerical and categorical parameters are supported. The algorithm randomly selects subsets of the training instances to compare configurations on. The size of this subset increases with each generation, so that better configurations are tested on more training instances to improve the accuracy of the comparison.

Add-ons

Smit and Eiben (2009) present two useful add-ons for tuning methods. Add-ons are "methods for increasing search efficiency that are independent from the main tuner and can be combined with different tuning algorithms". First, racing, as discussed before, can be used to reduce the number of expensive

measurements. To achieve this, configurations are measured on a variable, instead of fixed, number of problem instances. Statistically inferior configurations are only measured on a few problem instances, whereas promising configurations are measured on an iteratively increasing number of problem instances. Second, they propose sharpening, which aims at reducing the tuning time as well. At the beginning, sharpening uses only a small number of evaluations to determine the quality of a configuration. When a certain threshold is reached, a configuration is expected to be promising and the number of evaluations for this configuration is doubled. This ensures that the estimation of the quality of a configuration improves, since it reduces the effect of outliers. This idea is also used within SPO. In addition to the two mentioned add-ons, Dobslaw (2010) presents a third technique, adaptive capping, based on the ideas found in Hutter, Hoos, Leyton-Brown, and Stützle (2009). Its first variant, trajectory-persevering capping, corresponds to the notion of racing. Its second variant, aggressive capping, adds a time limit for evaluating configurations that bounds the allowed evaluation time based on the performance of the incumbent (i.e., the best so far) configuration.

Conclusion

From our discussion we conclude that the most promising ideas are found in the automated tuning methods. To compare the discussed methods, we assess each method on the features that are the most important in the light of our research:

Initial requirements We need a method that does not have many initial requirements (e.g., information about the parameter interactions and the performance landscape, since this is unknown beforehand).

Categorical parameters We need a method that is able to handle categorical parameters.

Set of instances We need a method that evaluates the performance on a set of problem instances.

Table 2.1 summarizes the features of the discussed automated tuning methods.

In the light of our research, the most promising ideas are found in the model-based SMAC and the model-free FocusedILS. To draw a final conclusion, we compare both methods in more detail. We discuss the qualitative differences first. FocusedILS needs a discrete set of parameter values for all parameters, including numerical parameters. This reduces the size of the solution space, but promising configurations might be disregarded. In fact, FocusedILS can only deal with categorical parameters. Since our numerical parameters are integers, we already have a discrete set for the numerical parameters, so our solution space is not narrowed. Additionally, the order of the training instances matters in the case of FocusedILS. However, choosing this order is not obvious, so an arbitrarily chosen order can negatively influence the performance. Second, quantitative differences exist. Hutter et al. (2011) discover that SMAC never performs worse than FocusedILS and that it yields statistically significant improvements. Moreover, since SMAC evaluates the performance without running the expensive algorithm itself, it needs less computation time. All in all, SMAC seems to be the most promising method in our case. Recall that SMAC using the random forests model of Breiman (2001).

To our knowledge, SMAC has never been applied to the VRP or a framework comparable to CVRS. Applying SMAC on these domains is therefore innovative. For this reason, it is likely that modifications of SMAC are needed to make it suitable for our case.

Depending on the time needed when we apply SMAC in our context, one of the discussed add-ons can be applied to further reduce the tuning time needed for a good performance. Racing has been successfully used in conjunction with SMAC by Styles and Hoos (2013). We come back to this in Section 5.3.5.

Method	Initial requirements	Categorical parameters	Set of instances
Jones et al. (1998)	−	− −	−
Bartz-Beielstein et al. (2005)	−	− −	−
Frazier et al. (2008)	−	+	−
Mes et al. (2011)	+	+	−
Birattari et al. (2002)	+	− −	+ +
Balaprakash et al. (2007)	+	−	+ +
Hutter et al. (2011)	+ +	+ +	+ +
Hansen and Ostermeier (2001)	+ +	− −	+ +
Nannen and Eiben (2006)	+ +	− −	+ +
Hutter, Hoos, Leyton-Brown, and Stützle (2009)	+ +	+ +	+ +
Ansótegui et al. (2009)	− −	+ +	+ +

Table 2.1: Features of automated tuning methods on a Likert scale (Likert, 1932).

Chapter 3

CVRS

In this chapter we explain how the optimizer CVRS works. This knowledge is needed in order to understand the search space, outlined in Chapter 4.

3.1 Terminology

In order to understand the working of CVRS, two key terms are important:

Task A task is defined as the transportation of a product (i) from a pickup location, or (ii) to a delivery location. Common properties of a tasks are the pickup or delivery location, time windows, and amounts quantifying the product of the transport.

Route A route is an ordered sequence of tasks, executed by one vehicle.

Given a set of tasks and a set of routes, CVRS aims at planning these tasks in those vehicles respecting the given restrictions (e.g., time windows) and optimizing the predefined criteria (e.g., driving distance).

The Vehicle Routing Problem is a NP hard problem. Therefore, it is unlikely that the problem can be solved to a guaranteed optimal solution in reasonable time, and thus a heuristic approach is used to solve the VRP.

3.2 Algorithms

We can roughly divide the algorithms used in CVRS in two phases: a construction phase and an improvement phase. The construction phase (see Section 4.2) constructs an initial solution and the improvement phase (see Section 4.3) tries to improve this initial solution. Since the construction algorithm used in CVRS does not necessarily find the optimal solution, the solution can often be improved in the improvement phase.

3.2.1 Construction

The construction phase of CVRS does not plan all tasks in one iteration. Instead, the tasks are divided in batches, which are planned in separate iterations. These batches contain, for example, tasks that are geographically close to each other. Different options can be specified for how to form those batches. Note that it is not mandatory to form batches.

After the division in batches, the cheapest insertion heuristic (see Section 2.1.2) is used to construct routes. The algorithm inserts the given set of tasks at the cheapest insertion point in the routes constructed so far. The algorithm divides all tasks in groups and keeps together clusters (e.g., a group

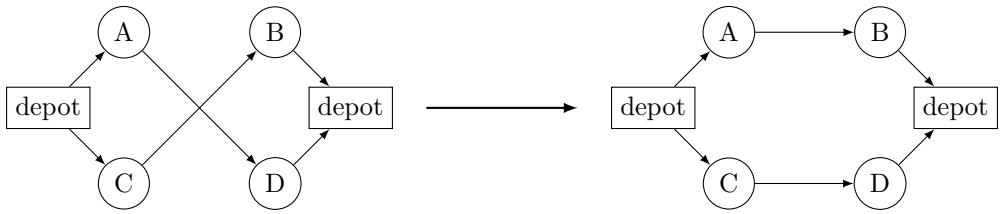

Figure 3.1: An example of the 2Opt algorithm.

of tasks having the same address). For a group, insertion points are found where the whole group is planned in a single route but parts of a group can be planned at different insertion points. The best overall combination of insertion points is found.

3.2.2 Improvement

The improvement phase is done after the construction phase of each batch, but for the entire solution. In this section we discuss the five possible improvement algorithms used in CVRS. Within the descriptions of the algorithms, often references to settings are made. An explanation of these settings can be found in Section 4.3.

The improvement algorithms are executed recursively: a certain sequence of improvement algorithms is executed several times consecutively. This is beneficial since some improvement algorithms create 'space' for other improvement algorithms to further improve the solution.

2Opt

The 2Opt algorithm tries to improve the current solution by means of removing crossings from the routes. This can be done within a route, but also between routes since usually it is not profitable if routes cross each other. In the CVRS implementation of the 2Opt algorithm an attempt is made to remove a crossing as long as the expected gain is higher than $MinimumEstimatedGain$ and after an improvement the algorithm is performed for at most $MaxNofIterations$ times. Moreover, in CVRS a tabu-list is used for the additional performance gain: if an option is tried and infeasible it will not be tried again until the corresponding route(s) has changed. An example of a change caused by the algorithm is visualized in Figure 3.1.

Swap

Swap tries to find a better solution than the current solution by swapping two (groups of) tasks in the solution. All combinations of groups are tried as long as the estimated gain is better than the setting $MinimumEstimatedGain$. The maximum number of times the process is repeated is set in $MaxNofIterations$. The process is only repeated if the solution has changed since the last iteration, including the creation of new groups based on the new solution.

CROSS exchange

The CROSS exchange algorithm tries to improve the current solution by means of resolving overlapping route parts between routes. This can be seen as performing two consecutive 2Opts, or a Swap with specific groups. An attempt is made to remove an overlapping route part as long as the estimated gain is higher than $MinimumEstimatedGain$. After an improvement, the algorithm is performed for at most $MaxNofIterations$ times. An example of a change caused by the algorithm is visualized in Figure 3.2.

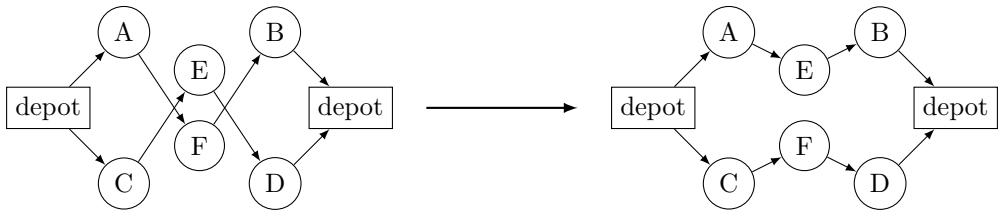

Figure 3.2: An example of the swap and CROSS exchange algorithm.

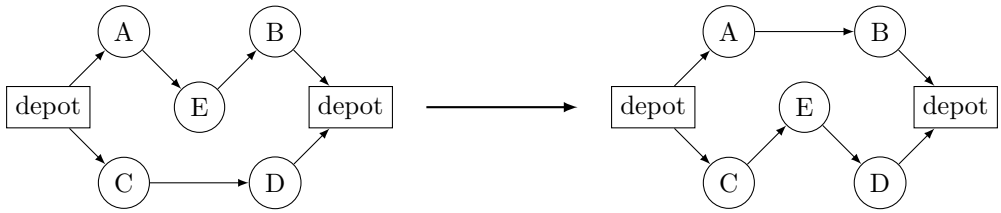

Figure 3.3: An example of the move algorithm.

Move

Move tries to find a better solution than the current solution by moving (groups of) tasks to another location. All combinations of groups are tried as long as the estimated gain is better than the setting $MinimumEstimatedGain$. The maximum number of times the process is repeated is set in $MaxNofIterations$. The process is only repeated if the solution has changed since the last iteration, including the creation of new groups based on the new solution. An example of a change caused by the algorithm is visualized in Figure 3.3.

LargeNeighborhoodMoveAndSwap

This version of the Move/Swap tries to find a better solution than the current one by moving (groups of) tasks to a different location in the solution. This is done by moving either one or two groups at once. All combinations of groups are tried as long as the estimated gain is better than the setting $MinimumEstimatedGain$. The maximum number of times the process is repeated is set in $MaxNofIterations$. This process is only repeated if the solution has changed since the last iteration, including the creation of new groups based on the new solution.

3.3 Solution quality

No uniform definition of the quality of a solution exists, as the objectives differ among customers. Therefore, we are able to define our own criteria to optimize, the optimization criteria. The optimization criteria are the targets for the calculations. This could be, for example, the number of tasks planned, the distance, or the number of routes used.

For this purpose, multiple objectives can be defined. The order of these objectives is important, since this defines the order of importance (known as the lexicographical order). That is, an objective is more important than all the subsequent objectives. For example, consider the following problem. We want to determine if a solution A is better than solution B. First, we compare the value of the first objective. If the value of solution A is better than the value of solution B, solution A is superior. In the case the values are equal, we measure the second objective in the same way. We iterate until a decision is made.

In the literature VRP problems typically have two objectives: the number of vehicles and thereafter the total distance (in lexicographical order as well). In literature problems it is common that all tasks

can be planned, that is, there is an abundance of available vehicles. However, in practice this is often not the case. Therefore the first objective in practical situations is typically the total number of planned tasks. Subsequently the same objectives as in the literature are used.

Chapter 4

Search space

As explained in Chapter 1, a configuration can be split in three levels. These three levels are found in the template of the default configuration. We explain this template in Section 4.1. We already introduced the search space of level 1, the improvement algorithms, in Section 3.2. In Section 4.2 we introduce the search space of level 0, the construction algorithm settings, and level 2, the improvement algorithms settings in Section 4.3. Next, Section 4.4 discusses the size of the search space. In Section 4.5 we analyze the currently used configurations to reveal a priori knowledge about the performance landscape. Finally, in Section 4.6, we explore the performance landscape in more detail, by conducting experiments.

4.1 Configuration template

CVRS comes with a default configuration. This configuration is hardly used in practice, because better performing configurations can be made (tailor-made). Nevertheless, the default configuration is often the starting point for those tailor-made configurations. This default configuration is used as a template. It determines the moments when we divide the tasks in batches, execute the construction phase, and execute the improvement phase. This template is often slightly different in tailor-made configurations. In principle, we stick to this template in our research. We could have included different templates by adding a categorical parameter to our search space that defines the template to use. However, adding the template to our search space (i.e., allowing different templates to be searched) would make our search space very irregular. It would therefore be better to run the tuning method several times, each time with a different template. Comparing the best configurations found in each run will lead to the best configuration.

4.2 Construction algorithm settings

The construction algorithm can be tuned by changing the values of its settings, called level 0. The construction algorithm does not plan all tasks in one iteration. Instead, the tasks are divided in batches first, which are subsequently planned in separate iterations. These batches might contain tasks that are, for example, close to each other, based on distance.

Currently, CVRS cannot create routes to plan transports on. Therefore, a set of so-called starting routes is input for the planning. Those starting routes are typically empty routes: they do not contain tasks yet. However, they already have a start location (e.g., the location the truck starts at) and contain information about capabilities (e.g., the route might be executed with a refrigerated truck). This makes it possible to use sorting criteria in the construction phase. Sorting criteria define how the tasks are

Sorting criterion	Direction
AngleClosestRouteToTask	Decreasing
DistanceClosestRouteToTask	Increasing
DistanceIncreaseFromBestToSecondBestRouteForTask	Decreasing
MaxEndOfRouteSlack	Increasing
SecondsToFinishOfTask	Increasing
SmallestDistanceToAnyAddressInSolution	Decreasing

Table 4.1: Common sorting criteria for the construction algorithm.

divided in batches. By specifying a sorting criterion, we steer which tasks are supposed to be together in a batch. This results in a list of tasks, sorted by the criterion. The first batch consists of the first N tasks in the sorted list, where N is a settings that specifies the maximum size of a batch. The next batch consists of the next N tasks in the sorted lists. This repeats until all tasks are part of a batch. Thereafter, the construction algorithm, cheapest insertion, is executed for all batches.

4.2.1 Sorting criteria

A plurality of options for the sorting criteria exists, about 50 in total. Because of performance reasons and specific customer wishes, only a few are used in practice. For all criteria a direction has to be configured. This specifies if a lower or higher value is favorable. However, the value of the direction is, in practice, fixed for each sorting criterion. Table 4.1 shows common choices for the sorting criteria, with their direction. For our experiments, we only use these common sorting criteria.

It is possible to first sort the tasks on the number of possible routes, in increasing order. A task cannot be planned in a route if, for example, the truck has insufficient capacity or a refrigerated truck is needed but the route is executed by a non-refrigerated truck. By sorting tasks on the number of possible routes, the algorithm is forced to plan those tasks first. This avoids the problem that might occur if these tasks would have been planned later: no possible routes might be available by then.

The following list gives an explanation of the common sorting criteria. Recall that we already have starting routes before we start constructing our initial solution (see Section 4.2). When we mention routes we refer to those starting routes or the routes constructed by then.

AngleClosestRouteToTask The angle from the closest route to the task. This sorting criterion likely results in routes that contain tasks that are geographically close to each other.

DistanceClosestRouteToTask The average distance from the closest route to the task. This sorting criterion likely results in routes that contain tasks that are geographically close to each other.

DistanceIncreaseFromBestToSecondBestRouteForTask The difference in distance between the closest route and the second closest route for the task. If this difference is large, it is likely that a large detour is needed if the task is not planned in its closest route. This sorting criterion likely results in routes that contain tasks that are geographically close to each other.

MaxEndOfRouteSlack The maximum amount of slack at the end of a route for possible routes of the task. If this slack is large, the route has still lots of room for additional tasks. This sorting criterion likely results in routes that take about the same time.

SecondsToFinishOfTask The number of seconds left to the finish of the task. If this number is small, the task is due soon and should therefore be planned soon. This sorting criterion is likely to make

Setting	Type	Range
OnlyAllowChangesWithinTheSameRoute	Categorical	True, False
MaxNofIterations	Integer	[0,200]
EstimateWith	Categorical	Costs, Distance, DrivingTime
MinimumEstimatedGain	Integer	*varies*[1]
MaxTravelTimeBetweenConsecutiveTasks	Integer	[0,1800]
OnlyModifyRoutesChangedInLastIteration	Categorical	True, False
MaxNofFailedInsertionsPerRoute	Integer	[0,10]

[1] The range of $MinimumEstimatedGain$ depends on the chosen value for $EstimateWith$, so a common range does not apply here.

Table 4.2: Improvement algorithm setting types and ranges. Note that the most common range is shown, and that slightly different ranges might apply to some improvement algorithms for performance reasons.

sure that tasks that are due soon are planned on time.

SmallestDistanceToAnyAddressInSolution The smallest distance from the task to any address already in the solution. This sorting criterion is likely to make sure that routes are geographically spread.

The sorting criteria operate as follows. Suppose that we choose to use $AngleClosestRouteToTask$ as our sorting criterion. For all tasks, we calculate the angle with the closest route. Based on these results, we create a list of tasks, sorted in decreasing order. The first N tasks are grouped into a batch, whereafter cheapest insertion is executed for this batch. The process iterates until all tasks are placed in batches, and planned with cheapest insertion. Note that we, more or less, mimic the sweep algorithm by Gillett and Miller (1974) (see Section 2.1.2) in this case.

4.2.2 Batch size

Next to the sorting criteria, we can also specify the maximum batch size N. An integer range of 1 to 20 seems appropriate here. Note that a batch size of 1 results in batches containing 1 task, which is equal to not dividing tasks in batches.

4.3 Improvement algorithms settings

Each improvement algorithm can be tuned by changing the values of its settings, called level 2. In this section we will discuss these settings. The settings that take a numerical (e.g., integer) value have, in theory, an infinite domain. However, from experience we can estimate a suitable range for each setting. For an overview of all settings with their type and range, we refer to Table 4.2.

Note that we already omit settings from which ORTEC knows that they do not perform well, in terms of improving the solution quality, in practical cases. We also leave out settings that are always customer-specific (i.e., they depend on the strategy of a business). This narrowing of our search space, is our first attempt to leave out settings that are not relevant. As a result, we can focus our tuning on the important settings, from which we expect that tuning significantly impacts performance.

The following list gives an explanation of the possible settings.

OnlyAllowChangesWithinTheSameRoute If this setting is true, the modifications are only done within the routes and not over multiple routes.

MaxNofIterations Maximum number of iterations done by the modification command. Regardless of this setting, each command stops whenever no improvement is found.

EstimateWith Unit in which the $MinimumEstimatedGain$ of a modification is measured. Currently the possibilities are: Distance, DrivingTime and Costs.

MinimumEstimatedGain An estimation of the gain due to the modification is made before the actual modification is done. This estimation is done based on entry for the $EstimateWith$ setting. The modification will only be done if this estimation is higher than the $MinimumEstimatedGain$.

MaxTravelTimeBetweenConsecutiveTasks If task B succeeds task A in the solution and task A is part of a group (see Section 3.2.1), then task B will only be added to that group if the driving time A to B is smaller than the maximum driving time in seconds provided in this setting.

OnlyModifyRoutesChangedInLastIteration If this setting is true, only the routes that are changed in the last iteration are candidates for modification in the new iteration.

MaxNofFailedInsertionsPerRoute If this setting is larger than 0, each group will only be tried in different places in a route for a maximum of $MaxNofFailedInsertionsPerRoute$ times.

The choice for $EstimateWith$ strongly affects the range for $MinimumEstimatedGain$ since it determines its unit. For example, a value of 1 for $MinimumEstimatedGain$ would, for $EstimateWith = Distance$, imply 1 kilometer, which is considered as a reasonable choice. However, the same value for $MinimumEstimatedGain$ would, if $EstimateWith = DrivingTime$, mean 1 seconds, which is too small in practical cases. Therefore, different ranges for $MinimumEstimatedGain$ are relevant, based on the value for $EstimateWith$. Nevertheless, these ranges have in common that all take about 10 values. That is why we use 10 as the number of possible values for $EstimateWith$.

The time an improvement algorithm needs to perform an iteration depends, apart from the values of its settings, on the algorithm it self. The swap algorithm is an expensive algorithm, meaning that an iteration takes a lot of time, compared to the other algorithms. This has implications for the range of values for its settings. For example, $MinumumEstimatedGain$ typically takes a higher value for expensive improvement algorithms. By this, the improvement algorithm is only executed once it is expected to yield a fairly large improvement. This, mostly, avoids the situation that the improvement algorithm is executed and no improvement is found (in those cases, the estimated gain is typically low as well). We take this into account with our tuning method.

4.4 Size

In this section we estimate the size of our search space, as this may foster the development of a suitable tuning method.

4.4.1 Construction phase

For the construction phase, we first have the option to sort the tasks on the number of possible routes or not (so 2 options). Regardless of this choice, we thereafter have to choose a sorting criterion from the list of 6 options. Moreover, we have 20 options for the size of a batch. This results in a total number of $2 \times 6 \times 20 = 240$ options for the configuration algorithm.

Settings	Possible values
OnlyAllowChangesWithinTheSameRoute	2
MinimumEstimatedGain	10
MaxNofIterations	201
EstimateWith	3
MaxTravelTimeBetweenConsecutiveTasks	1801
OnlyModifyRoutesChangedInLastIteration	2
MaxNofFailedInsertionsPerRoute	11
Total	4.8×10^8

Table 4.3: Total size of the search space per improvement algorithm. Since we have 5 different improvement algorithms, the total search space for one place in the sequence of improvement algorithms is $4.8 \times 10^8 \times 5 \approx 2.4 \times 10^9$.

4.4.2 Improvement phase

In theory, the length of the sequence of improvement algorithms could be infinite. In practice, this is definitely not recommended since the time needed to solve an instance would exceed all limits by far (see Section 1.3.2). Therefore we limit the size of this sequence in the configuration to 10. That is, we allow a *maximum* of 10 improvement algorithms, executed one after another, to improve the initial solution. Given the length of the sequences in the current configurations and the estimated computation time required with 10 improvement algorithms, this seems to be appropriate. Note that this sequence is executed recursively (see Section 3.2.2).

Based on Table 4.2, we can determine a rough estimate of the total size of our search space for the improvement phase. Table 4.3 shows the result.

Our sequence consists of 10 places and since the improvement algorithms are executed in the given order, the order of the improvement algorithms matters. As a result, we have 10^{10} possible sequences of improvement algorithms. Combined, this results in a total number of $2.4 \times 10^9 \times 10^{10} \approx 2.4 \times 10^{19}$ possible improvement phase configurations.

4.4.3 Total size

Since we do not want to ignore possible interaction effects between construction and improvement settings, we consider the whole search space at once. We obtain a total number of $240 \times (2.4 \times 10^{19}) = 5.8 \times 10^{22}$ configurations. As 5 minutes is a rough lower bound for the time needed to evaluate one configuration on one instance, it would take billions of years to explore the complete search space. Complete enumeration is therefore not an option.

Moreover, if we limit the time allowed for tuning to a weekend, we can only measure about 1000 configurations, which is only a small fraction of $\frac{1000}{5.8 \times 10^{22}}$ of the search space. This justifies our choice for offline tuning, as measuring one configuration already approaches the online computation time limit of 15 minutes. That is, during online tuning we can only measure one configuration. Obviously, multiple measurements are needed for tuning and therefore offline tuning should be used.

From our analysis, we conclude that our search space is finite. That is, there is a bounded number of possible configurations since no parameter takes continuous values.

4.4.4 Limiting the size

As we showed in Section 4.4.3, the size of our search space is very, very large. In order to efficiently spend our tuning time, we could limit the size of the search space beforehand, if we can limit the risk that we lose many promising configurations.

Although all configurations in the discussed search space are unique, minor changes in the configuration are not likely to have high impact on the performance of the configuration. For example, changing the value of $MaxTravelTimeBetweenConsecutiveTasks$ from 313 to 314, can be considered as such a minor change. If we do not take these minor changes into account, we reduce our search space, which is intuitively likely to accelerate our tuning method (in other words, explore the search space more efficiently), without creating a high risk that we lose many promising configurations. However, Hutter et al. (2011) performed experiments with SMAC on both a discretized search space and a full (i.e., continuous) search space. In their case, much better results were achieved on the full search space. Hence, we do not limit our search space by discretizing settings. Note that SMAC will not evaluate all possible values for a certain parameter one by one, but it will evaluate the values based on larger intervals first.

Although excluding some parameter values is not advisable, excluding some parameters might be beneficial. By excluding parameters with a small effect on the solution quality, we focus on the most important part of the search space. By analyzing the performance landscape, we reveal this information (see Section 4.6).

4.5 A priori knowledge

Many customers use the optimizer CVRS on a daily basis. The command templates (i.e., algorithm configurations) used, are tailor-made by ORTEC's consultants and developers. Therefore, these command templates comprise a lot of, potentially, useful information. It would be wise to take this information into account, as it is gathered mainly by experience. In this section we try to extract this information, in such a way that we can use it in our tuning method.

4.5.1 Analysis

We analyze a set of 10 command templates that are currently used in practice. The customers of these command templates differ, also in their type, allowing us to find information that applies in general, and not for a specific customer.

OnlyAllowChangesWithinTheSameRoute

A pattern that we find in many configurations is to first only look at intra-route changes, and afterwards to inter-route changes (i.e., changing $OnlyAllowChangesWithinTheSameRoute$ from true to false). Obviously, there are less options to move a task within a route than between all routes, and because of this, calculating intra-route changes is cheaper. This way, we force the algorithm to look at intra-route improvements first. As a result, improvements can be found in an inexpensive way since only one route is involved. Later on, if this inexpensive way is not likely to find improvements anymore, the algorithm looks for possibilities involving more than one route. Since this choice is based on performance, and the impact on the solution quality is not known for sure, we do not implement it in our method.

EstimateWith

For each improvement algorithm, we have to specify the value of $EstimateWith$. Although different values are used in different configurations, we notice that this setting always has the same value at

all places in a configuration. Recall that this setting is used to compute an estimation of the gain for a change, and that the modification is executed when the estimation is promising. CVRS uses a configurable objective function, so the definition of gain depends on this. For example, if in the objective function costs are considered as more important than distance or driving time, the use of costs as a definition for the gain is evident. The value of $EstimateWith$ depends therefore largely on the objective function. As a result, we fix the value of $EstimateWith$ to a value that is logic in relation with the objective function used.

MaxTravelTimeBetweenConsecutiveTasks

Generally speaking, the value of $MaxTravelTimeBetweenConsecutiveTasks$ decreases for improvement algorithms, as we move to the end of the sequence. The idea behind this approach is in fact the same as for $OnlyAllowChangesWithinTheSameRoute$: first look for improvements in an inexpensive way, and use expensive ways later on. We do not implement this in our method, for the same reason as for $OnlyAllowChangesWithinTheSameRoute$.

4.5.2 Conclusion

We were not able to gather lots of information from the command templates, since the command templates differed much mutually. This reaffirms that tuning of the algorithm is customer-specific.

4.6 Performance landscape

In this section, we analyze the performance landscape by performing experiments with several configurations. By this, we gather relevant information that supports the development of a tuning method.

In order to find useful information about our search space we have to draw samples from the large search space of configurations, since measuring all configurations would take too much time. To draw relevant conclusions based on this sample, we have to make sure that this sample is representative for the whole search space. A common technique to achieve this is Latin Hypercube Sampling (LHS). LHS first divides the range of each parameter into n intervals that do not overlap and have an equal size. Next, it random selects one value from every interval for every parameter. Finally, these selected values are randomly combined with the selected values of the other parameters.

Using our experiments, we want to draw conclusions on the effect of the different levels of a configuration. For each level, we use LHS to draw 50 configurations where only the parameters related to that level vary. Hence, in total we will have (3 levels $\times$ 50 configurations per level =) 150 drawn configurations. The other parameters are fixed at their default values. We perform measurements with the 150 configurations on $instance_A$. We define the effect of a level on an objective as the difference between the average performance of the 50 configurations of the corresponding level and the average performance over all 150 configurations, divided by the average performance over all 150 configurations. This allows us to draw conclusions about levels relative to the other levels. Note that a lower effect represents a better performance, as it is common in tuning literature to minimize the objectives.

Figure 4.1 shows our results. We notice that none of the levels had an effect on the number of tasks planned. Apparently, there are enough vehicles available to be able to always plan all tasks. If we look at the number of vehicles used, we notice that level 0 had the best impact. From this we conclude that the construction phase is largely responsible for the performance gains that can be made in total, since the objectives are lexicographical ordered. Note that solely tuning level 0 increases the total distance about 15%, so each vehicle has to cover much more distance. Level 1 and level 2 clearly have the best

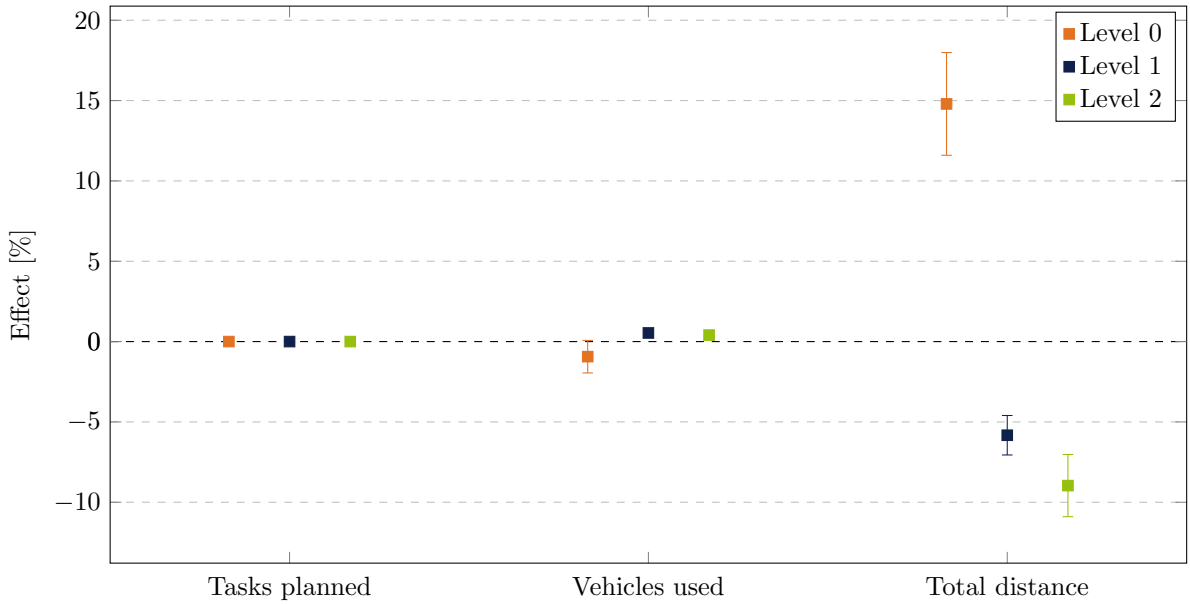

Figure 4.1: The effect per level for the various objectives with their 95% confidence interval. A negative effect represents a better performance.

effect on the total distance. The improvement phase, consisting of both level 1 and level 2, is therefore useful as well.

From our analysis we conclude that all three levels have a significant effect on the solution. Hence, we take all three levels into consideration when tuning.

Furthermore, we notice that the drawn configurations have an effect of only a few percent on the performance on the first two objectives. This leads to the conclusion that it is either hard or impossible to find configurations that perform better than the best randomly drawn configuration. We previously compared performances with the average performance over all 150 drawn configurations, to compare the levels mutually. To further investigate this, we now compare the results of the 150 configurations with the performance of the default configuration on $instance_A$. The measurement with the default configuration results in a performance that is about equal to the average performance over the 150 configurations. Hence, a comparison with the default configuration results in about the same performance as shown in Figure 4.1. From experience we know that in many cases larger improvements over the default configuration are possible. We therefore conclude that, at least in this case, good performing configurations are hard to find (a simple LHS is not sufficient) and that a good tuning method will be very useful.

In Section 2.2.2, we predicted that our search space is irregular. We indeed notice this if we take a detailed look at the performances of the 150 configurations. We illustrate this with an example. We consider two configurations, denoted by θ_A and θ_B, that differ only in the chosen sorting criterion. Although the solution is equal in the number of tasks planned, θ_A uses 7% less vehicles than θ_B but the total distance is 27% more. From this we conclude that a small change (e.g., only the sorting criterion) may result in significant performance differences.

Chapter 5

Tuning method

In Chapter 2 we concluded that SMAC is likely to be a suitable method. Our tuning method, which we outline in this chapter, is largely based on the ideas of SMAC.

First, in Section 5.1, we explain in detail how to overcome the two issues addressed in Section 2.2.2 that make common tuning methods unsuitable: categorical parameters and a set of instances. Second, in Section 5.2 we perform an initial experiment with the standard version of SMAC. Finally, in Section 5.3 we present our tuning method more precisely.

5.1 Tackling general problems

In Section 2.2.2, we addressed two issues that make many tuning methods inapplicable to our problem: (i) the occurrence of categorical parameters, and (ii) solution quality that should be measured on a set of instances. In Section 5.1.1 we tackle the first problem, the latter in Section 5.1.2.

5.1.1 Categorical parameters

Typically, tuning methods are limited to numerical parameters. In order to handle categorical parameters, of which many exists in our search space, we use a model based on random forests (Breiman, 2001). Random forests are collections of regression trees, which have the performance of the algorithm, the solution quality, at their leaves. An example of a random forest with 2 trees can be found in Figure 5.1 (the random forests we use consist of 10 trees, see Section 5.3.3). A regression tree is a decision tree with numbers at the leaves, rather than class labels. Regression trees are obtained by recursively partitioning training vectors into smaller subsets, and the (average) performance of the vectors in the subset is a prediction for vectors similar to the vectors in the subset. Regression trees are able to handle categorical data (as shown in Figure 5.1) and they are known to perform well in such cases (Bartz-Beielstein & Markon, 2004). Breiman (2001) shows that random forests result in more accurate predictions, caused by using multiple trees rather than only one tree. Random forests allow to quickly predict the performance of a configuration, without having to execute an expensive measurement. By this, we are able to efficiently choose the configurations we measure, which is very time-consuming.

To build a random forest, we use a data set with the solution quality of all configurations measured so far on instances of the training set. Using this, we start constructing the trees. At each node of a tree, we take a random subset R of the configuration parameters to be eligible to be split upon. Hence, the parameters in the subset differ among the trees in the forest. Moreover, the parameters in the subset are different each time we build our random forest. The size of this random subset R is pre-determined and always smaller than the total number of parameters (a common choice is to take $\frac{5}{6}$ of the total number of

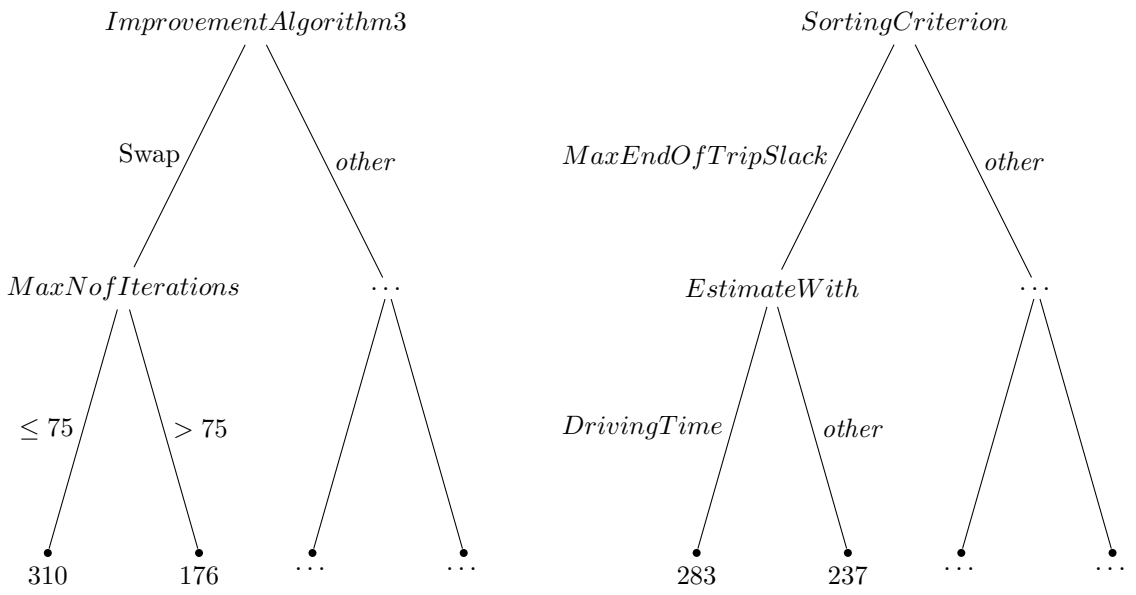

Figure 5.1: An example of a random forest with 2 trees.

parameters). From the subset of parameters R, we randomly select a parameter r and a threshold value t for it. We create two groups of configurations based on this parameter and threshold value: group A contains the configurations for which the value of parameter r is smaller than or equal to t, and group B contains all other configurations. For categorical parameters, group A contains the configurations for which the value of parameter r is equal to t, and group B contains all other configurations. For each group, we again randomly select a parameter from $R \setminus r$ and a threshold value. We repeat the same process. Note that the groups contain increasingly less configurations, since their are split each time. We only allow a split on a certain parameter, if the number of data points in that node exceeds a certain threshold. This prevents us from predicting performance based on only a few data points, which is likely to be inaccurate. If $R = \emptyset$ or if a group contains insufficient configurations, we stop branching there: we arrive at a leaf. Since we never branch on all parameters in a tree, we are able to create a forest that consists of multiple, distinct trees.

We are now able to predict the performance of a configuration, by propagating the configuration down each of the trees, and taking the mean of the data points in the leafs we end up in. This is a very inexpensive way of predicting the performance, as opposed to the expensive way of measuring the performance with the (routing) algorithm.

The information we gather by successive configuration measurements, new data points, is added to the trees, which makes subsequent predictions more accurate.

In Section 5.3.3 we explain when, and how, we use a random forest in our tuning method.

5.1.2 Set of instances

Tuning methods typically try to tune parameters based on one instance of a problem. The resulting parameter set is then suitable for the given instance. However, we have to find a set of parameters that performs well for future, unseen instances, and we use a set of training instances that is representative for those instances. As a consequence, the tuning method should be able to cope with this. This becomes an issue when assessing the performance of a configuration θ_{new}: we have to decide on how many training instances π we have to measure in order to conclude that a configuration θ_{new} is the new best configuration. This best configuration so far is denoted by the incumbent configuration θ_{inc}. This

mechanism is generally referred to as the intensification mechanism.

At the beginning of the intensification mechanism of SMAC an attempt is made to improve the accuracy of the prediction for the performance of our current θ_{inc}. Since we have only measured θ_{inc} on some instances of the training set, we are not sure if the current prediction is representative for the complete training set. Hence, we first measure θ_{inc} on a randomly selected instance on which θ_{inc} has not been measured yet, to improve the quality of our prediction. Next, we perform measurements with θ_{new} until we either reject it or accept is as our new θ_{inc} (an assessment we outline later in this section).

```
input  : list of configurations Θ_new, incumbent configuration θ_inc, training instances Π, and time
         bound t_intensify
output: incumbent configuration θ_inc, and updated measurement data MeasurementsData

for i = 1,...,length(Θ_new) do
    θ_new ← Θ_new[i];
    π' ← random instance from Π where θ_inc has not been evaluated on;
    if π' is not null then
        MeasurementsData ← Measure(θ_inc,π');
    end
    NofMeasurements ← 1;
    while true do
        for i = 1,...,NofMeasurements do
            π' ← random instance from Π where θ_inc has been evaluated on, but θ_new not;
            if π' is not null then
                MeasurementsData ← Measure(θ_new,π');
                Π_common ← instances for which we have measured both θ_inc and θ_new;
            else
                if performance(θ_new,Π_common) is not worse than performance(θ_inc,Π_common) then
                    θ_inc ← θ_new;
                    break;
                end
            end
        end
        if performance(θ_new,Π_common) is worse than performance(θ_inc,Π_common) then
            break;
        end
        NofMeasurements ← 2 × NofMeasurements
    end
    if time spent > t_intensify and i ≥ 2 then
        break;
    end
end
```

Algorithm 1: The intensification mechanism of SMAC (Hutter et al., 2011).

In Section 1.1, we noted that the routing algorithm is deterministic. As a consequence, there is no need to measure a configuration multiple times on the same problem instance. However, we choose instances to measure on randomly to ensure that the way the instances are ordered has no influence. By this, we add a form of stochasticity. Running the tuning method twice could therefore result in different outcomes. To limit the effect of stochasticity, we measure θ_{new} on an instance that is previously solved by θ_{inc}. By this, we ensure that we make a fair comparison between θ_{inc} and θ_{new}. Otherwise, if we compare the performance of θ_{inc} and θ_{new} on different instances, differences in the instances (i.e., the optimal solution quality, in absolute terms, could differ largely) influence the comparison.

A list of configurations is taken as an input for the intensification mechanism. This list contains two types of configurations: (i) configurations marked as promising and (ii) randomly sampled configu-

rations. Randomly sampled configuration are included to ensure that SMAC converges to the optimal configuration (see Section 5.3.1). This list alternates between both, that is, a promising configuration is followed by a randomly sampled configuration and vice versa. Once θ_{new} is rejected or becomes the new θ_{inc} and there is still time left for intensifying (a predefined amount of time), the next configuration from the list is selected, and the same procedure is executed.

If θ_{new} performs better on the instance than θ_{inc}, we increase the number of measurements for θ_{new} using a doubling scheme. That is, we double the number of instances to measure θ_{new} on. By this, we improve the quality of the prediction for θ_{new}, which decreases the probability that we accept bad configurations as θ_{inc}. Note that this idea is similar to the principle of sharpening (see Section 2.2.2). We repeat until either θ_{new} performs worse (we reject θ_{new}), or we have measured θ_{new} on as many instances as θ_{inc} (θ_{new} becomes our new θ_{inc}). Note that when we compare performances, we always look at the performances on the common instances Π_{common}: the instances for which we have measured both θ_{inc} and θ_{new}.

Algorithm 1 shows our intensification mechanism. In Section 5.3.5 we explain when, and how, we use the intensification mechanism in our tuning method.

5.2 Initial experiment

To assess the performance of SMAC, and to gather information that might be useful for the development of our tuning method (we might have to adapt SMAC), we first conduct an experiment with SMAC, with the settings as proposed by Hutter et al. (2011). We compare SMAC with another recent tuning method: uRace (Van Dijk, 2014).

Van Dijk (2014) used uRace to tune the parameter settings, the configuration, of an algorithm for solving loading instances. In these instances, a set of items should be placed in a set of containers. However, the container capacity is insufficient to accommodate all items, and the objective is therefore to choose the best subset of items. To measure the quality of a solution, the percentage of a container that is occupied, the volume utility, is used.

Since SMAC contains some random elements (e.g., the instances to measure a configuration on), we perform multiple runs, each with a different seed value. When we discuss performance, we discuss the average performance over these runs. We start each run with an empty forest, that is, we do not use data gathered in previous runs. We set the time bound for the intensification mechanism to 5 seconds.

We consider an experiment where we tune the parameters to optimize the performance on the set of 700 loading instances used by Van Dijk (2014). As a training set, we take a subset of this complete set. We allow a similar amount of time for tuning as Van Dijk (2014) used in this experiment. Tuning by Van Dijk (2014) resulted in a volume utility of 83.99 on the complete set; our experiment resulted in a volume utility of 84.10 ± 0.28 (95% confidence interval). Van Dijk (2014) unfortunately does not report the 95% confidence interval for these experiments. Nevertheless, we conclude that SMAC gives at least comparable results as uRace (and maybe slightly better).

Solving an loading instance used by Van Dijk (2014) takes, at most, about 10 seconds. In our routing case, this takes up to 20 minutes. Therefore, we compare thhe time spent on measuring the performance of configurations during the tuning process, to be able to estimate the performance of SMAC on our routing case. Note that the other time (the total tuning time minus the time spent on measuring the performance of configurations) is spent on generating and predicting the performance of new configurations. The time spent on measuring configurations was more than 99% of the tuning time with uRace, but with SMAC approximately 87% of its time. Given that the total tuning time was equal, we conclude that uRace conducted about 10% more measurements. There is no obvious reason

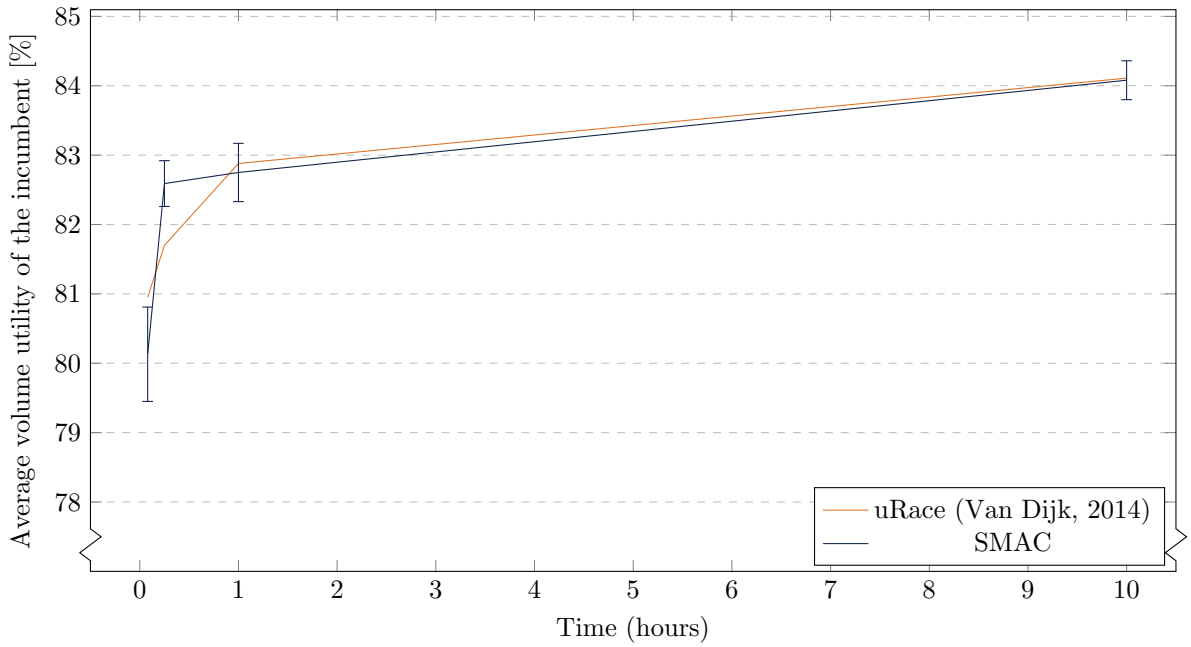

Figure 5.2: The volume utility of the incumbent configuration over time. For SMAC, the average volume utility with its 95% confidence interval is shown. Note that the volume utility might be measured on (a part of) the training set, due to the intensification mechanism. To make a fair comparison with uRace, we only use the volume utility after 5, 15, 60 and 600 minutes as data points, since these are reported by Van Dijk (2014).

that the time needed to generate new configurations increases significantly more for SMAC than for uRace in other experiments. Therefore, SMAC is likely to perform better in cases with a larger time per measurement.

In our routing case we have relatively (i.e., the total tuning time relative to the time per measurement) less time to tune. Therefore, we need a method that finds good configurations fast. Figure 5.2 shows that SMAC finds configurations with a significantly better volume utility faster than uRace. For our routing case, SMAC seems therefore to be the preferred method. However, during the tuning process, time is also needed to generate and predict the performance of new configurations. This time differs between the tuning methods. We already observed that SMAC, in total, needs less measurements than uRace. In our routing case, we limit our tuning time to a weekend (see Section 1.3.2). The significant difference in time needed for a measurement between the loading and routing case means that we are able to handle significantly less measurements for routing. To assess the benefit of SMAC, it is important to know if the number of measurements made is less than when using uRace. Since random forests need to be filled with some measurement data, SMAC might need many measurements at the beginning. Figure 5.3 shows that SMAC makes less measurements than uRace per time unit, immediately from the start of tuning. During the first hour, SMAC makes 17% less measurements than uRace. The benefit of SMAC over uRace is therefore present, as it makes fewer measurements, given the same tuning time, which becomes a real benefit if measurements are expensive. We previously noted that, in total, about 10% less measurements were needed. From this, we conclude that SMAC makes more measurements per time unit later on in the tuning process. This is caused by our intensification mechanism. After a while, the incumbent configuration has been measured on many instances. To determine if a new configuration is a new incumbent configuration, the new configuration is measured on many instances as well (if it is not worse). Therefore, many measurements are scheduled, leaving fewer time for generating and predicting new configurations.

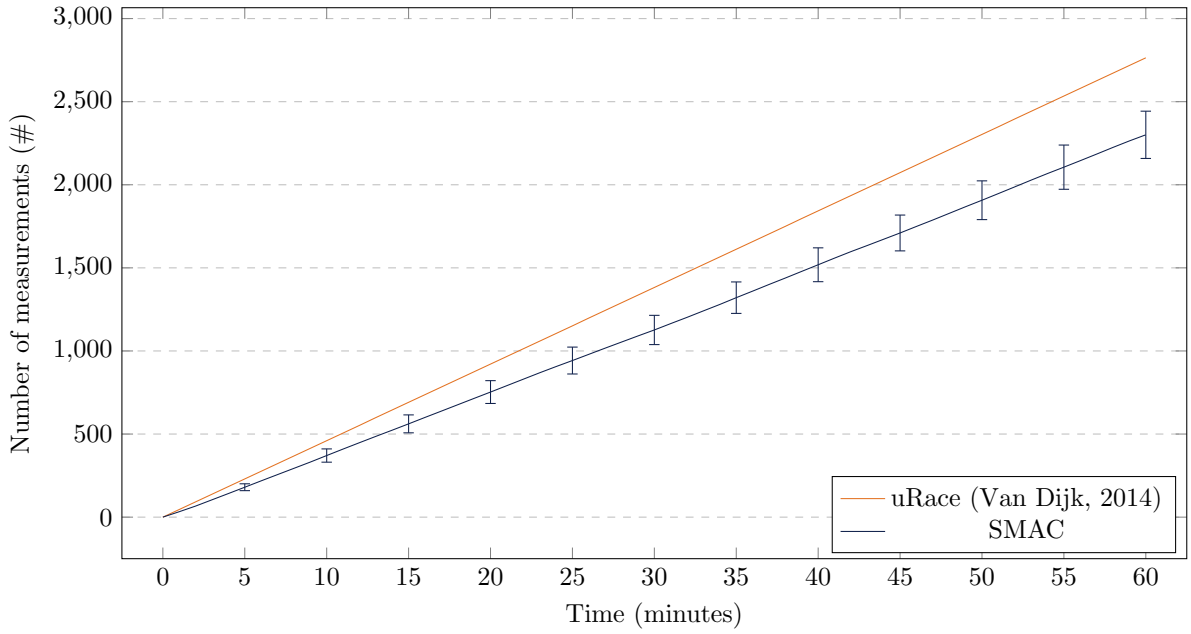

Figure 5.3: The number of measurements during the first hour. For SMAC, the average number of measurements with its 95% confidence interval is shown. Note that Van Dijk (2014) did not report these numbers, but we calculated them using the information reported by Van Dijk (2014).

From this experiment, we conclude that SMAC is a well performing method, and that it is likely to be suitable for cases in which an measurement is expensive (e.g., in our routing case).

5.2.1 Comparison

Since both uRace and SMAC have now been tested under exactly the same conditions, we can compare both methods in detail. This may lead to interesting insights that we can use for the development of our tuning method.

uRace starts with a set of configurations, sampled using LHS. Based on this set, from which configurations are removed once they are inferior, new configurations are generated. To find inferior configurations, a statistical test is used. However, for this test, a lot of measurement data is needed: all sampled configurations should be evaluated on some instances, before a configuration can be marked as inferior. In our routing case, where a measurement takes minutes, it would take hours to eliminate a substantial part of the set of configurations. However, experienced users could already predict that some configurations in this set are inferior: they could have eliminated a substantial part already before measuring them. SMAC, on the other hand, starts with a single configuration, typically the default configuration. From this configuration is known that it performs reasonable. No time is therefore wasted on eliminating configurations for which we could already predict that they were inferior.

To find new promising configurations, uRace uses its set with non-inferior configurations. A well-performing configuration from this set is selected, and another configuration evolves from this one that lies in its neighborhood. This process results in a single configuration that is measured afterwards. However, it would not be uncommon that the performance of this new configuration, is worse than the configuration it evolved from, since the performance landscape is irregular. The chance that this occurs when using SMAC is much lower, since the expected performance of a new configuration is, quickly, predicted before measuring it (see Section 5.3.4). These inferior configurations could therefore be detected in an inexpensive way.

5.3 Structure of our tuning method

In this section we outline our tuning method. In Section 5.3.1 we proof that SMAC converges to the optimal configuration. Algorithm 2 shows the general structure, unchanged from SMAC, in which different phases exist. First, we start with the initializing of a configuration (in Section 5.3.2). Second, we build the random forest (in Section 5.3.3). Third, we generate new configurations that are likely to perform well (in Section 5.3.4). Fourth, we assess the performance of the most promising configuration, to find out if it performs better than the current incumbent configuration (in Section 5.3.5). In this phase we perform new measurements which yield new measurements data for our random forest. Afterwards, we again build a random forest, including the newly gathered data, and repeat subsequent steps until we run out of time. We denote the process of building the forest, generating promising configurations and assessing the performance as an iteration. Finally we address practical issues we have to take into account: the limited computation time in Section 5.3.6 and the multiple objectives in Section 5.3.7.

input : algorithm A, configuration search space Θ, training instances Π, and initial configuration θ_{inc}
output: incumbent configuration θ_{inc}

Initialize(θ_{inc});
while *tuning time left* **do**
  $RF \leftarrow$ BuildRandomForest($MeasurementsData$);
  $\theta_{new} \leftarrow$ GeneratePromisingConfiguration(θ_{inc},RF);
  [θ_{inc},$MeasurementsData$] $\leftarrow$ Intensification(θ_{new},θ_{inc},Π);
end

Algorithm 2: The tuning approach (adapted from Hutter et al., 2011).

5.3.1 Convergence

In Section 4.4.3 we concluded that our search space is finite. In this section we provide the proof that SMAC theoretically converges to the optimal configuration in such search spaces. This proof is taken from Hutter et al. (2011), but we give it here for completeness and discuss it in our context (e.g., our search space and measuring performances with CVRS). This proof indicates the boundaries we have to take into account if we want to make changes to SMAC, but still want to ensure that the method converges to the optimal configuration. Once we make a change to SMAC that affects this proof, we explicitly mention it.

First, we give a definition of a consistent estimator. A concept we use later on to proof the convergence. In the definition we distinguish the empirical performance and the true performance: the empirical performance is the performance obtained by measuring the configuration a number of times on several instances and the true performance is the performance across all possible, relevant instances (e.g., also future, unseen instances).

Definition 1 (Consistent estimator)**.** *$\hat{c}_N(\theta)$ is a consistent estimator for $c(\theta)$ iff*
$\forall \epsilon > 0 : \lim_{N\to\infty} P(|\hat{c}_N(\theta) - c(\theta)| < \epsilon) = 1$

With $\hat{c}_N(\theta)$ we denote the empirical cost (i.e., the inverse of the solution quality) of configuration θ based on N measurements. If $\hat{c}_N(\theta)$ is a consistent estimator of $c(\theta)$, the estimate of the true cost becomes increasingly reliable as N approaches infinity. Eventually, mistakes in comparing two configurations are eliminated.

Lemma 2 (No mistakes for $N \to \infty$)**.** *Let $\theta_1, \theta_2 \in \Theta$ be any two parameter configurations with $c(\theta_1) < c(\theta_2)$. Then, for consistent estimators $\hat{c}_N, \lim_{N\to\infty} P(\hat{c}_N(\theta_1) \geq \hat{c}_N(\theta_2)) = 0$.*

For a proof of this lemma we refer to Hutter, Hoos, Leyton-Brown, and Stützle (2009).

Lemma 3 (Unbounded number of measurements)**.** *Let $N(I,\theta)$ denote the number of runs SMAC has performed with parameter configuration θ at the end of iteration I. Then, if the maximum number of runs for the incumbent configuration is set to ∞, $\lim_{I\to\infty} P[N(I,\theta) \geq K] = 1$ for any constant K and configuration θ in Θ (with finite Θ).*

Proof. The intensification mechanism used in SMAC (see Algorithm 1) makes sure that in each run of the intensification mechanism always at least one randomly sampled configuration is measured. Therefore, each configuration has a positive chance of being measured in each iteration. With a probability of $p = 1/|\Theta|$ this is configuration θ. Thus, the lower-bound for the number of measurements for configuration θ is a binomial random variable $B(k; I; p)$. Therefore, $\lim_{I\to\infty} B(k; I; p) = \lim_{I\to\infty} \binom{I}{k} p^1 (k-p)^{I-k} = 0$ for any $k < K$. Hence, $\lim_{I\to\infty} P[N(I,\theta) \geq K] = 1$. Since the routing algorithm of CVRS is deterministic (see Section 1.1), we only need one measurement for each configuration per instance (K=1). □

Theorem 4. *When SMAC operates in a finite configuration space Θ and uses a consistent estimator $\hat{c}_N$ for cost measure c, the probability that it finds the true optimal configuration approaches one as the time allowed for tuning goes to infinity.*

Proof. Each iteration takes finite time. Hence, if the time allowed for tuning goes to infinity, the number of iterations I goes to infinity as well. Lemma 3 states that if I goes to infinity, the number of measurements for each configuration θ grows unboundedly. According to Lemma 2, the true best configuration will be selected in a comparison if the number of measurements for the configurations goes to infinity. With probability arbitrarily close to one, SMAC will eventually measure all configurations and will select the true best configuration over all other configurations. □

According to Hutter, Hoos, Leyton-Brown, and Stützle (2009), cost estimators are often not consistent in practical cases. Since SMAC uses a training set with instances, the cost estimators can only estimate the performance on this set: performance on future, unseen instances cannot be measured. Even for a large N, the estimate is only accurate for the instances in the training set. Especially if the size of the training set is small, the difference between the empirical and true performance might be large. To account for this, the instances of the training set should be representative for the future, unseen instances. Instead of using a finite training set, using a distribution over relevant problem instances might partly solve these issues. However, determining this distribution might be difficult to achieve in practice. A practical solution is to use large training sets, with instances that are representative of the distribution. A large training set minimizes the expected difference between the empirical and true performance.

In our case we use a finite training set with instances that are as representative as possible for future, unseen instances. Nevertheless, we cannot be completely sure that the instances are indeed representative. As a consequence, we cannot be complectly sure that SMAC converges to the optimal configuration in our case. We try to limit the difference between empirical and true performance by (i) choosing relevant instances and (ii) taking a fairly large training set. We discuss this in Section 6.2.

5.3.2 Initial incumbent configuration

The initial incumbent configuration, is the configuration we start our tuning process with. We have 2 obvious options: (i) the default configuration, or (ii) a randomly sampled configuration. Given the enormous size of our search space, we could give ourselves a head start by using the default configuration, since it is not likely that a randomly sampled configuration performs as good as the default configuration. If we would opt for the randomly sampled configuration, we would have to spend quite some of our tuning time to find a configuration that performs as a good as the default configuration.

Hence, we start our tuning method with a measurement of the default configuration, on a randomly chosen instance from the training set. This configuration remains the incumbent configuration, until we have found a better performing configuration.

5.3.3 Building the forest

After we measured the initial configuration, or after we measured a promising configuration, we build a random forest using the data we gathered with measuring the configurations so far. Our random forest consist of 10 trees. We build our trees using the procedure described in Section 5.1.1.

5.3.4 Generating a promising configuration

We generate a new promising configuration (i) if we find a new incumbent configuration, or (ii) if a previously promising configuration is rejected (i.e., we stop evaluating the configuration since we do not expect it to be better than our current incumbent configuration). We define a promising configuration as a configuration that has a high likelihood of performing better than the incumbent solution. To measure how promising a configuration is, we calculate its expected improvement (EI). The EI is not obtained by directly evaluating the performance, but by calculating an expectation of the performance using a simple, less-expensive model. We already addressed the concept of expected improvement in Section 2.2.2, when discussing EGO (Jones et al., 1998).

To generate new promising configurations, we have to make sure that we do not end up in local optima. Therefore, we present a method that generates both configurations in known good parts of the search space (exploitation), and configurations in unknown parts of the search space (exploration). We deal with this, by making sure that the value of EI is high for both configurations with high predicted performance, and for those with high uncertainty (i.e., configurations in those parts of the search space where we do not know the performance landscape yet).

The use of random forests that was introduced as a solution to deal with categorical parameters, has another major advantage. In Section 1.2 we already outlined that we need a method that severely limits the number of expensive measurements. A random forest allows us to quickly compute the predictive μ_θ and σ_θ for a configuration θ. We use this prediction, rather than a performance measurement, to find promising configurations. As a consequence, less expensive performance measurements are needed. These predictions become more accurate, as the number of data points used for the random forest increases (i.e., when the tuning method runs for a while). These predictions are a cheap substitute for the expensive performance measurements.

In SMAC, expected improvement is defined as

$$EI(\theta) := f_{min}\Phi(v) - e^{\frac{1}{2}\sigma_\theta^2+\mu_\theta} \cdot \Phi(v - \sigma_\theta), \tag{5.1}$$

where $v := \frac{\ln(f_{min}) - \mu_\theta}{\sigma_\theta}$, Φ denotes the cumulative distribution function of a standard normal distribution, and f_{min} denotes the empirical mean performance of θ_{inc}, see Hutter et al. (2011). We use the logarithmic transformation of the empirical mean performance f_{min} in v. By this, we optimize $\max\{0, f_{min} - e^{f(\theta)}\}$. Hutter, Hoos, Leyton-Brown, and Murphy (2009) show that this logarithmic transformation increases the predictive accuracy. For more details, we refer to Hutter, Hoos, Leyton-Brown, and Murphy (2009).

To gather new promising configurations, we follow the local search procedure on the most promising, previously measured, configurations from Hutter et al. (2011). In more detail, we first calculate the EI for all previously measured configurations using Equation 5.1. Next, we categorize the ten configurations with the highest EI as promising. For the local search, we use a randomized one-exchange neighborhood, including four random neighbors for each numerical parameter, and configurations that differ on one

discrete parameter value. This search procedure is very local, which has the advantage that we take parameter interactions into account. These interactions are important in our case, since it is likely that a particular parameter setting might behave very different if another parameter setting changes: for example, $MaxNofIterations$ (level 3) might perform very different, if another improvement strategy (level 2) is chosen. For the EI, computations for a batch of N configurations are much cheaper than N separate computations (Hutter et al., 2011), and therefore we measure all neighbors at once. Once all of the neighbors have a smaller or equal EI than the best EI so far, the local search procedure stops. From the resulting list, a list with unseen configurations and their EI, we select the configuration with the highest EI, as the new configuration θ_{new} to measure.

For our initial experiment (see Section 5.2), we also computed the EI of 500 randomly sampled configurations[1], since these batch computations are cheap, and might lead to promising configurations. If so, we use this configuration as our new configuration θ_{new} to measure. However, we noticed that we never selected a randomly sampled configuration; in all cases the configurations that evolved from the local search had a larger EI. To spend our tuning time more efficient, we leave out randomly sampled configurations in subsequent experiments.

In Hutter et al. (2011) always at least one promising configuration and one randomly sampled configuration are measured in each run of the intensification mechanism. This to make sure that SMAC converges to the optimal configuration (see Section 5.3.1). Now we propose to leave out randomly sampled configurations, we lose this characteristic: it is no longer sure that every configuration is measured at least once. Nevertheless, it is very unlikely that we find the optimal configuration in our case anyway. This because it would require much more time than we allow for tuning in order to be sure that SMAC converges to the optimal configuration. However, the EI is also high for configurations in unexplored parts of the search space, since the uncertainty is high for those configurations. It is therefore likely that every configuration will be labeled as promising eventually. As a result, every configuration has a high likelihood of being predicted using the random forest.

5.3.5 Assessing the promising configuration

After a new, promising configuration is generated, we have to assess it in order to conclude if it is indeed performing well. To this end, we use an adapted version of the intensification mechanism described in Section 5.1.2. This results in an incumbent configuration, either the same as before or the generated promising configuration. If we have tuning time left at this moment, we build a new random forest (using the additional data gathered in the intensification procedure), and repeat the other phases subsequently.

Our algorithm shows a structural change, compared to the intensification mechanism originally proposed in Hutter et al. (2011) (see Algorithm 1). In their case, a list of configurations was taken as an input. We propose to use a single configuration for the following reason. Since we need up to 20 minutes for a performance measurement, we have to allow a substantial amount of time for intensifying, to allow multiple configurations to be measured per run of the intensification mechanism. We believe that we can spend this time more efficiently by generating a new promising configuration first, using the newly gathered data, and intensifying that configuration afterwards. This because generating a new promising configuration is not expensive, and it could well lead to a configuration that is better than the other configurations in the list.

In addition, we want to improve the quality of the configuration we measure since measuring is expensive. In particular, we want to limit the number of measurements for bad configurations. Hence, we have to determine as early as possible if a configuration is bad or still promising. In Section 2.2.2 we

[1] Hutter et al. (2011) proposed to use 10000, but this takes minutes of the tuning time (likely due to the complexity of the search space, where many conditional parameters exist), which we found undesirable

discussed three add-ons that can be used for this purpose: racing, aggressive capping, and sharpening. Currently, configurations are compared using the intensification mechanism we described in Section 5.1.2.

First, we could use a principle similar to racing. For example, we could use a statistical test to determine if a configuration is better than another configuration. For a typical statistical test, at least 5 common instances should be measured by both configurations in order to draw a valid, useful conclusion. However, 5 common measurements is already very expensive in our case, given that a single measurement already takes minutes. Moreover, if the performance on a single instance already differs much between two configurations, we can already stop the measurements for one of the two configurations. This is because ORTEC does not want a configuration that performs really bad on a certain instance (even if it would do reasonably good on average). Therefore, we conclude that it is likely that the principle of racing does not limit the number of measurements for bad configurations sufficiently rigorous.

Second, aggressive capping adds a time limit for measuring configurations. This add-on is only effective if the objective of tuning is to minimize the runtime of the configuration. Since our aim is the solution quality instead of the runtime, we cannot apply this technique.

Third, sharpening increases the number of measurements for a configuration as long as the configuration looks promising. The ideas found in this technique are currently used by SMAC (e.g., the doubling scheme used to increase the number of measurements for the incumbent). However, we believe that we can modify the implementation of the technique in SMAC such that it performs better in cases with a large time per measurement (as in our case). We propose the following two modifications:

Increasing the number of measurements The standard version of SMAC uses a doubling scheme to increase the number of measurements N for a new configuration. Recall that we compare a new configuration only after these N configurations, and not in the meantime. However, it might be the case that we can already see after less than N measurements that the new configuration performs worse. Especially if the time needed for a measurement is large, it would be beneficial to compare earlier so that we can reject the new configuration already. By this, we would not waste our valuable time on this poor configuration any longer, but we can spend this time on measuring new promising configurations. To achieve this, we propose to always increase the number of measurements N with 2, instead of multiplying it with 2 (as the original doubling scheme does). Our proposal makes sure that we compare new configurations earlier with the incumbent, by which we can reject them earlier if needed. Moreover, for each run of the original intensification mechanism, a measurement with the incumbent configuration is performed first to increase the accuracy of the performance of the incumbent configuration on the complete training set. Recall that we always compare performances on the common instances. Therefore, the increased accuracy is only needed once we also have to measure the new configuration on that many instances. We therefore propose to increase the number of measurements for the incumbent configuration only when needed. That is, when the number of measurements N is larger than the number of measurements we already performed for the incumbent configuration. We increase the number of measurements for the incumbent if we find a configuration that does not perform worse than the incumbent configuration on the common instances until then.

Comparisons based on one common instance The first comparison for a new configuration with the incumbent configuration is based on only one instance. The standard version of SMAC already stops with measurements for the new configuration if it performs worse than the incumbent configuration on this single instance. However, it is likely that the optimal configuration does not perform best on all instances individually. Therefore, rejecting a configuration based on the performance of one instance may lead to a faulty rejection. On the other hand, if we measure each configuration on more than one instance before comparing it to the incumbent, we severely restrict the number

of different configurations we can measure in a given amount of time. Moreover, if a configuration performs really bad on the first instance, we do indeed want to reject it since we do not want a configuration that performs really bad on some instance(s). Measuring each configuration on more than instance at any rate seems therefore inappropriate. To deal with this, we propose to correct the performance of a new configuration when comparing it with the incumbent based on one common instance. In detail, we allow a small deterioration (e.g., 1%) of the performance in this case. By this we make sure that we reject configurations that perform really bad, but we do not reject configurations that perform almost as good as the current incumbent configuration.

Solely based on theoretical knowledge, we are not able to identify the most promising modification(s). We therefore investigate the effect of all modifications using experiments in Section 6.1 after which we are able to select the best modification(s).

5.3.6 Limited computation time

In Section 1.3.2 we stated that we do not allow configurations that result in a computation time exceeding 15 minutes. However, our tuning method as described so far, might result in a configuration that exceeds 15 minutes. Instead of having to run the tuning method again with slightly different settings or instances, which would hopefully result in an acceptable configuration, we build in a mechanism to prevent this.

Once we measure a configuration, we record the time needed. If this exceeds our limit of 15 minutes, we make sure the algorithm does not make it its incumbent, and we store the time limit overrun in the respective leaves of the trees. This information is then used for future performance predictions as well: the prediction for configurations that end up in such a leave is that they will not finish in time. As a consequence, the probability that such configurations are labeled as promising decreases.

5.3.7 Multiple objectives

As discussed in Section 3.3, routing problem instances, and practical instances in particular, typically have multiple objectives. Hutter et al. (2011) did not address this, and SMAC cannot deal with this natively. We could considered to use multiple random forests, one for each objective, to take this into account. This implies that a configuration has multiple EI values. In case of a tie in EI for some configurations, the next EI (representing the next objective) should be used. We would have to build a random forest for each objective to achieve this. Based on the results of our initial experiment, it seems likely that this would take a considerable amount of time. Therefore, we opt for a simplified way to take multiple objectives into account, in which we do not have to build multiple forests.

The idea behind the approach is to translate the n multiple objectives' values into a single value. To do this, we have to estimate, for each objective i (except the last one), a number M that is sufficiently larger than the largest possible value for the next objective (lexicographical sorted): $M_i \gg \max(value_{i+1})$. To calculate the single value, we have to be aware that some objectives have to be minimized and that others have to be maximized. We convert all objectives to objectives we have to minimize, by using M. The price of an objective

$$price_i = \begin{cases} M_{i-1} - value_i, & \text{for objectives that have to be maximized,} \\ value_i, & \text{for objectives that have to be minimized.} \end{cases}$$

The single objective value that has to be minimized is computed as $\prod_{i=1}^{n} M_i - \sum_{i=1}^{n} (\prod_{i=0}^{n-i} M_{n-i}) \cdot price_i$. This value can be used as performance measure, since it is a single value for the multiple objectives' solution quality.

Chapter 6

Experimental results

In this chapter we discuss the experimental results of our research. In Section 6.1, we analyze the effect of the modifications we proposed in Section 5.3.5. In Section 6.2, we discuss the design of our training and test set. In Section 6.3, we assess the performance of our tuning method followed by the performance of our tuned configurations in Section 6.4. Finally, in Section 6.5, we discuss the versatility of our tuning method.

6.1 Modifications

We introduced several modifications to SMAC in Section 5.3.5 that might make the tuning method perform better in our case. We were not yet able to identify the most promising modification. To our knowledge, the proposed modifications have never been applied before to SMAC or similar methods. As a consequence, we analyze the modifications using experiments. For these experiments we want to limit the time needed for practical reasons, but we want to make sure that we can draw valid conclusions afterwards. We therefore use routing instances that are not too difficult and can be solved in less than a minute by CVRS. The use of simpler instances makes it likely that we do not find large differences between the modifications, since many configurations will find the same solution. As a consequence, above-average solutions are rare and hard to find so small differences indicate outstanding performance. Using the experiments we executed in Section 4.6 with an instance of one of ORTEC's customers, we revealed a similar performance landscape. Hence, we expect that we can draw conclusions based on the simpler instances that are valid for the customers' instances as well. Nevertheless, we recommended to further investigate if the behavior of the modifications is the same if they are applied directly to customers' instances.

Both our training and test set consist of the same type of instances. In subsequent experiments we use another design for the training and test set (see Section 6.2). However, since we want to focus on the performance of the modifications here, we use two sets that contain comparable instances.

We conduct experiments with five different versions, based on the modifications introduced in Section 5.3.5. We have two options for modification regarding increasing the number of measurements: multiply with 2 (denoted by M) or add 2 (denoted by A). We combine each of these options with the two options for the modification regarding comparisons based on one common instance: using a correction if there is only one common instance (denoted by C) or not using this correction (denoted by N). This results in four versions: $M + N$, $M + C$, $A + N$, and $A + C$. In addition, we also conduct experiments with the standard version of SMAC, i.e., without modifications (denoted by 'None' from now on). We implement the two modifications that use a correction if there is only one common instance as follows. Once we compare the incumbent configuration with a new configuration based on one common instance,

Modification	Average number of tasks (#)	# Best
None	60.58	42
$M + N$ (Multiply with 2)	60.64	43
$M + C$ (Multiply with 2 with correction)	60.70	45
$A + N$ (Add 2)	60.68	44
$A + C$ (Add 2 with correction)	60.48	40

Table 6.1: The performance of modifications on test set. By way of comparison, the default configuration of CVRS planned on average 59.00 tasks. The last column shows the number of instances on which the modification results in the best performance in terms of tasks planned. Because of many draws, this column adds to more than 50.

we continue with measurements for the new configuration only if the number of tasks planned is at least as good as for the incumbent configuration. That is, we do not allow a deterioration of our first objective (number of tasks planned), but do allow a deterioration of subsequent objectives.

Since our tuning method makes some choices randomly (e.g., which instance to measure on) that influence the outcome, we perform 3 experiments for each modification.

Table 6.1 shows the results of our experiments. The modification where we multiply the number of measurements with 2 and use a correction if there is only one common instances ($M + C$) seems to perform best, in terms of both average number of tasks planned and number of instances for which it performs best. The observed differences are small, especially between $M + C$ and $A + N$. The 95% confidence interval for the difference between the means of $M + C$ and $A + N$ is -0.02 ± 0.13. Since this interval includes zero, the difference is insignificant under $\alpha = 0.05$. We nevertheless use $M + C$ for subsequent experiments, since we already expected that the differences would be small and that small differences would indicate outstanding performance. Note that the performance on subsequent objectives is of less importance, since the objectives are lexicographical ordered and we have a difference on the first objective.

Algorithm 3 shows the intensification mechanism for the selected modification in detail. This is the intensification mechanism we use in subsequent experiments.

6.2 Training and test set design

In Section 5.3.1 we addressed two issues related to the design of the training set: (i) making sure that the training instances are relevant and (ii) taking a fairly large training set. At first sight it is not clear how to solves the issues in our practical case. We therefore discuss the design of the training and test set in this section.

To find suitable sets, we have to perform multiple experiments. For practical reasons we allow less than a weekend for tuning per experiment, which is not a problem since we consider these experiments as preliminary. As in the experiment of Section 6.1, we again perform multiple experiments per training set size. For these experiments we use the objectives as stated in Section 3.3: the number of planned tasks, the number of vehicles, and the total distance.

We use $instance_A$ for the experiments in this section, since we only have one instance in this case which should be used for both our training and test set. We will use the results found using this instance in Section 6.3.1 and Section 6.4.1. Since we have multiple instances of $instances_B$, we can use a less advanced design for our training and test set in the case of $instance_B$ (see Section 6.4.2).

```
input  : promising configuration θ_new, incumbent configuration θ_inc, and training instances Π
output: incumbent configuration θ_inc, and updated measurement data MeasurementsData

IncreasedRunsForIncumbent ← false;
NofMeasurements ← 1;
while true do
    for i = 1,...,NofMeasurements do
        π' ← random instance from Π where θ_inc has been evaluated on, but θ_new not;
        if π' is not null then
            MeasurementsData ← Measure(θ_new,π');
        else
            IncreasedRunsForIncumbent ← true;
            π' ← random instance from Π where θ_inc has not been evaluated on;
            if π' is not null then
                MeasurementsData ← Measure(θ_inc,π');
                MeasurementsData ← Measure(θ_new,π');
            end
        end
    end
    Π_common ← instances for which we have measured both θ_inc and θ_new;
    if corrected performance(θ_new,Π_common) is not worse than performance(θ_inc,Π_common) then
        if IncreasedRunsForIncumbent then
            θ_inc ← θ_new;
            break;
        else
            NofMeasurements ← NofMeasurements * 2;
        end
    else
        break;
    end
end
```

Algorithm 3: Our intensification mechanism (adapted from Hutter et al., 2011). With 'corrected performance' we denote that we correct the performance in case there is only one common instance for the comparison (as explained in Section 5.3.5).

6.2.1 Relevant training instances

Choosing relevant training instances brings us immediately to a practical problem that ORTEC faces. Customers using CVRS typically plan their tasks for a day or a few days ahead. The data involved in this is then what we call an instance. As a consequence, a week brings us only a few, large instances. However, we cannot collect instances during weeks before generating a suitable configuration: customers want this as soon as possible. Therefore we use a method to generate training instances from $instance_A$. We use the $instance_A$ as our test instance since this instance reflects a true instance. When generating training instances we have to make sure that they are relevant for $instance_A$ to avoid huge differences between the performance on the training and test set. For $instances_B$ we do not need to generate (additional) training instances, since we already have access to a set of instances. Hence, we can split this set to obtain our training and test instances.

To explain the method we use to generate training instances, consider the following example. Suppose we have our VRP $instance_A$ (see Section 1.3.1). The tasks have been planned in the 23 vehicles using some configuration. We now have a dataset containing 23 vehicles with some tasks for each vehicle. We can now take a subset of the vehicles (with their tasks) as a training instance. For example, we take a subset with 3 vehicles of $instance_A$ with their tasks as a training instance. We have to make sure that these training instances remain relevant for the original instance $instance_A$. To find out which subset

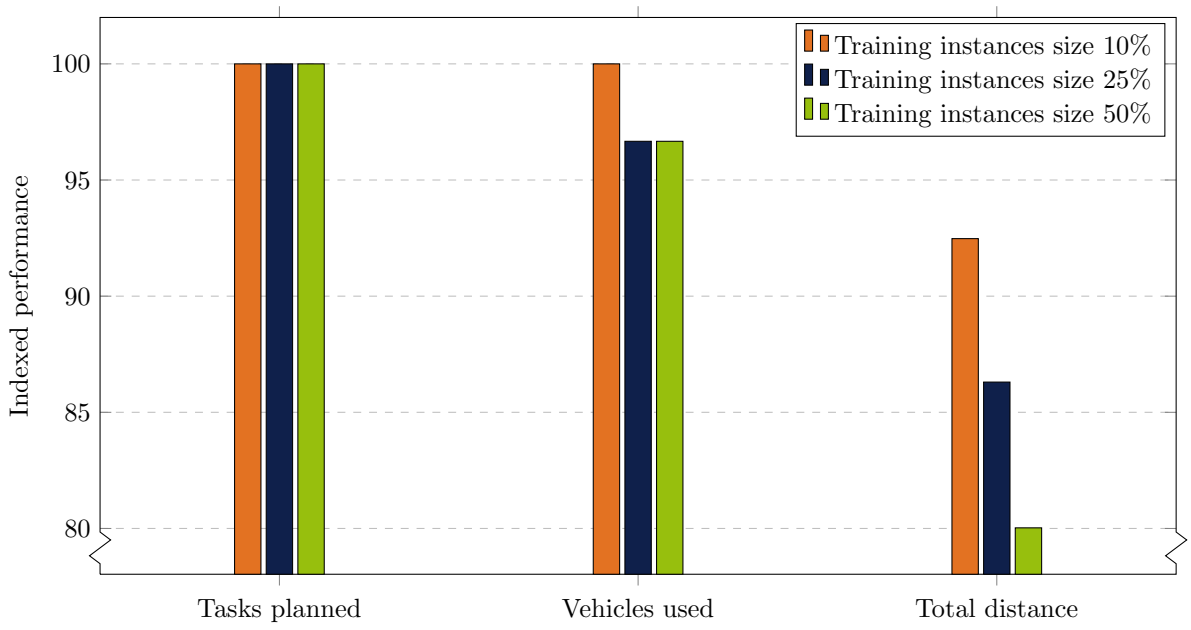

Figure 6.1: The indexed performance per training instance size for the various objectives on the test instance. An indexed performance of 100 represents the score of the default configuration. A lower indexed performance represents better performance.

size we should take, we perform experiments with different subset sizes. We consider experiments with instances containing 3, 8 and 13 vehicles of $instance_A$.

Figure 6.1 shows our results. The size of the subset does, apparently, not affect the performance on the first objective. We notice a difference on the second objective: training instances consisting of 3 vehicles of $instance_A$ perform worse than the other two. If we look at the third objective, we notice that tuning with the largest training instances results in the best performs. For subsequent experiments we therefore take instances consisting of 13 vehicles of $instance_A$[1]. That is, we generate training instances by taking 13 vehicles with their tasks from the set of 23 vehicles. By this, we can generate $\binom{23}{13} = 1144066$ training instances.

6.2.2 Training set size

The large number of possible training instances brings us to the issue of the size of the training set. In Section 5.3.1 we drew the conclusion that using a large set is advisable, since this minimizes the expected difference between the empirical and true performance. However, a large training set might also introduce adverse effects. For example, the intensification mechanism might result in a measurement of all instances in the training set. Although we already greatly reduced this possibility with our new intensification mechanism as opposed to the original mechanism, the possibility still exists. Measuring all training instances is undesirable since it takes a lot of time but it is not likely that is has a great effect on performance (i.e., after a while, the performance of the incumbent is likely to be stable). On the other hand, small training sets have the problem outlined in Section 5.3.1: the difference between the empirical performance on the training set and true performance might be large.

A suitable size for our training set is unknown so far. We perform experiments with different sizes of the training set to find a suitable size. As already discussed, our intensification mechanism restricts the

[1]This size of the training instances makes sure that CVRS needs minutes to solve an instance. By this, we test our tuning method for a case in which computation times are large. Since we developed a tuning method suitable for such cases, this seems suitable.

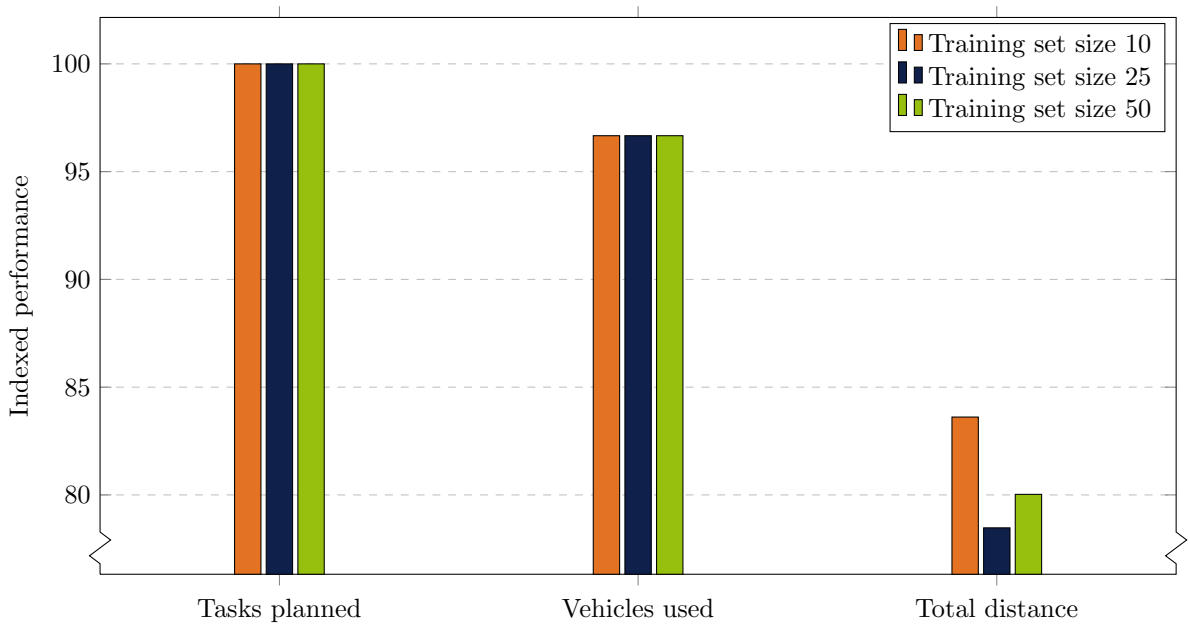

Figure 6.2: The indexed performance per training set size for the various objectives on the test instance. An indexed performance of 100 represents the score of the default configuration. A lower indexed performance represents better performance.

number of training instances measured. As a consequence, we do not have to analyze very large training sets. We do not expect the mechanism to measure the performance of a configuration on more than 50 instances. For our experiments, we use training sets with a size of 10, 25 and 50 training instances respectively.

In Section 5.3.6 we discussed how we are going to take to maximum computation time of 15 minutes into account. Note that this time limit holds for the true instance in the test set. Nevertheless, we have to take this into account when tuning on the training instances to avoid that we end up with a configuration that takes too long on the test instance. Since training instances are approximately half the size of the test instance, a time limit of 5 minutes would be intuitive. However, from experience we know that the structure of the instance has more influence on the required time than the number of vehicles or tasks. We therefore perform a small experiment in which we analyze the difference in computation time between the training instances and the test instance for a given configuration. This experiment shows that a configuration needs about half the time for a training instance as for the test instance. From this we conclude that a time limit of 7.5 minutes seems appropriate, since the computation time for the test instance is 15 minutes. Moreover, it shows that the structure of our training instances and test instance exhibit similarity. This is a first auspicious observation that our method to generate training instances seems suitable.

Figure 6.2 shows our results. What stands out is that the performance on the first two objectives is not affected by the size of the training set. The size of the training set seems therefore not as important as we expected. It might very well be the case that configurations performing better on the first two objectives are rare or even nonexistent. Nevertheless, we notice that a training set with 25 instances performs the best, since it has a better performance on the third objective. For subsequent experiments we therefore take training sets consisting of 25 instances. We do not take the same 25 instances for each experiment, but we take a different subset from the 50 training instances we generated. Moreover, this experiment shows that using larger training sets does not always result in increased performance. In this case, using more training instances causes SMAC to evaluate a configuration on more instances before rejecting it

or making it the incumbent configuration, instead of measuring other promising configurations. Hence, less different configurations are measured, which decreases the probability that it finds a very good configuration.

6.2.3 Test set

In the experiments of Section 6.2.1 and Section 6.2.2 we used the original instance $instance_A$ as our test instance since this instance reflects a true instance. Since the training instances were generated from this original instance, the training instances contain the same, but less, vehicles and tasks as the original instance. In practical situations this is not the case: based on previous instances we want to perform well on future, unseen instances. Hence, we have to separate training data from test data. However, we do not always have access to multiple true instances (e.g., in the case of $instance_A$). In such cases we cannot consider an instance as unseen (i.e., not take it into account when training, but only use it as test instance). Nevertheless, we still want to find out if a configuration is robust (i.e., performing comparable on two similar instances). As explained before, we consider instances from the same customer as similar.

To assess if the performance of a configuration is robust we perform the following experiment. Five times we randomly split our $instance_A$ into two halves, where each halve contains about the same number of vehicles (and tasks). We consider the 10 resulting instances as true instances for the same customer. Note that they have about half the size of a true customer instance. Next, we randomly sample 50 configurations from our search space. We measure the performance of each configuration on two randomly chosen instances (but two instances that belong together, i.e., that together make our $instance_A$). Afterwards, we calculate the indexed performance for a configuration on an instance by dividing the performance of the configuration by the average performance of all configurations. The performance indicator for this experiment is the absolute difference between the indexed performance of a configuration on its first and second instance.

For the number of tasks planned the difference is on average 0.22% with a maximum of 1.82%. These results immediately show that the performance of a configuration on both halves of $instance_A$ is robust on the most important objective. The average difference for the number of vehicles used is 9.46%. Although this may seem high at first sight, there is an explanation for this. The maximum difference for the number of vehicles used between a configuration and the average of all configurations is only 1 vehicle, the smallest possible difference except 0. Due to the small number of vehicles used (at most 9) the percentage is rather high. Nevertheless, the absolute difference is only small, which again indicates a robust performance. The last objective, the total distance, has an average difference of 5.51%. In previous experiments we already noticed that this objective fluctuated the most, and hence we may consider 5.51% as robust.

In subsequent experiments we use the original, true instance as our test instance. It is preferred to use another true instance as our test instance, but we do not always have access to this. Since we showed that the performance on two true instances is likely to be robust, it is also very likely that the result on our test instance are valid for other true, unseen instances as well. In subsequent experiments where we have access to multiple true instances (and hence use multiple true instances in our test set), we analyze the robustness of the performance on the test set.

6.3 Performance of our tuning method

In this section, we assess the performance of our tuning method by comparing it with a straightforward tuning method: random sampling. For random sampling we allocate the same amount of tuning time and employ the same set of training instances. The configuration to measure is sampled randomly as well as

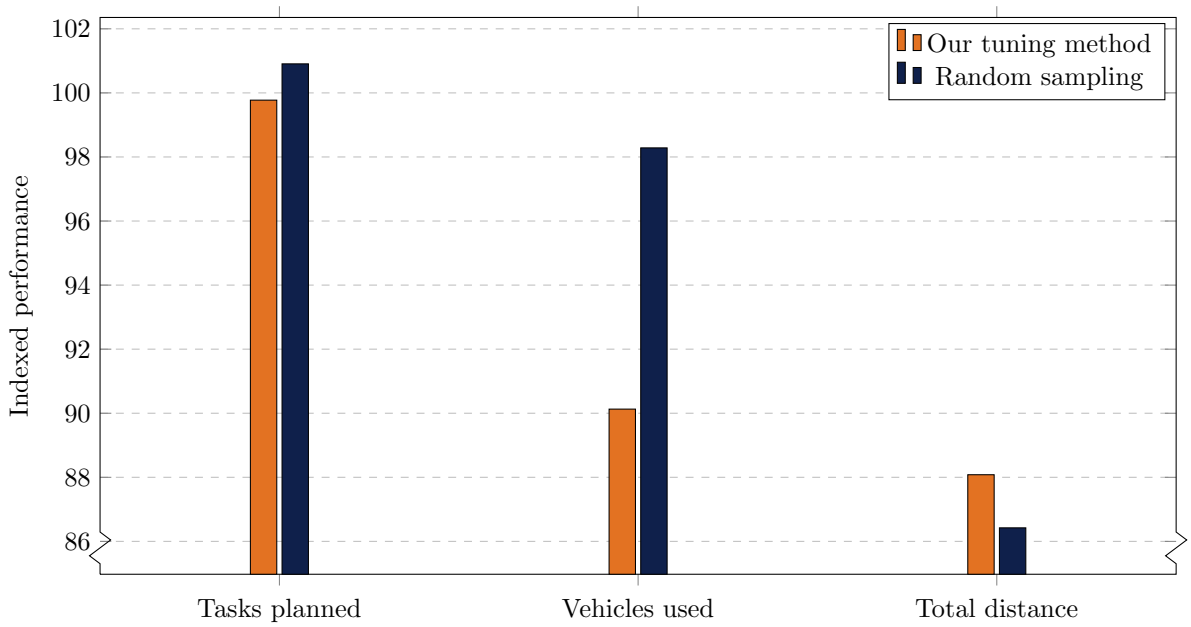

Figure 6.3: The indexed performance per tuning method for the various objectives on the training set of $instance_A$. An indexed performance of 100 represents the score of the default configuration. A lower indexed performance represents better performance.

the number of instances to measure a configuration on. By this, the choice that are made intelligently by our tuning method are made at random. Recall that we already compared SMAC to the tuning method uRace in Section 5.2.

6.3.1 $Instance_A$

For $instance_A$ we tune our configuration based on a training set that consists of 25 instances, whereby each instance contains 13 vehicles. To measure the performance of the tuned configurations, we use $instance_A$. This design is the best design for $instance_A$ according to our research in Section 6.2. For $instances_B$ we tune our configurations based on a training set that consists of the instances of 3 consecutive days. As a test set we use 2 instances that are from the 2 days consecutive to the training instances. Hence, we assume that we are at day 3 and want to find a good configuration for days 4 and 5. For the tuning method, days 4 and 5 are unseen instances as they are not used in the training set.

Figure 6.3 shows the results for $instance_A$. The difference on the first objective is small yet beneficial, but the second objective shows that our tuning method is significantly better than random sampling. Note that random sampling is worse than the default configuration. From this we conclude that it is beneficial to use our tuning method instead of random sampling.

Robustness

For ORTEC it is also important that a configuration is robust: the performance in terms of improvement over the default configuration should be positive for almost all instances. ORTEC does not want a configuration that performs very good on some instances, but also very bad on other instances. To investigate this, we consider the performance of the best found configuration on the training set. As we see in Figure 6.3, the improvement over the default configuration for the first objective is negligible. Moreover, from the results we see that the number of tasks planned on all instances, except one, is the same for both the default and the tuned configuration. Hence, we consider the second objective to analyze the robustness.

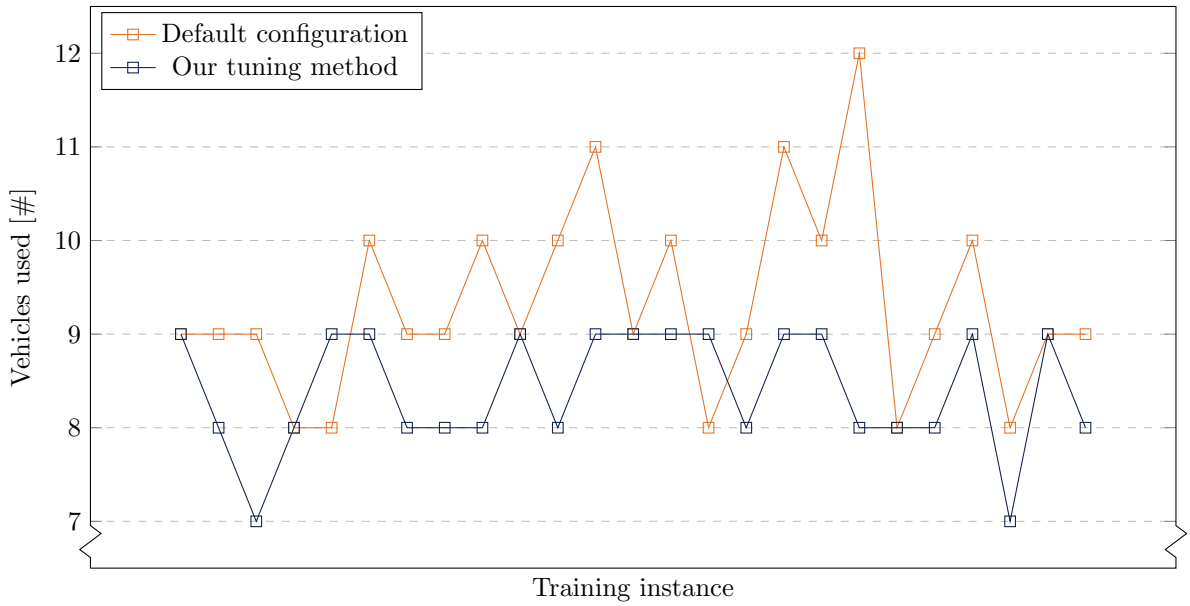

Figure 6.4: The performance on the 25 training instances for $instance_A$, for both the default configuration and our tuned configuration.

Figure 6.4 shows the performance on the second objective for all 25 training instances, both with the default configuration and our tuned configuration. There are only 2 instances on which our tuned configuration performs worse, but on these instances the difference is only 1 vehicle. From this figure we conclude that for the majority of the instances, the tuned configuration performs better than the default configuration. Hence, the configuration tuned with our tuning method is indeed robust.

6.3.2 $Instances_B$

For $instances_B$, ORTEC has a tailor-made configuration that performs very well on these specific instances: it has been made especially for $instances_B$. The tailor-made configuration uses a slightly different template (see Section 4.1) than used for the default configuration. To be able to make fair comparisons, we also use the template of the tailor-made configuration for the other tuning methods. That is, we use a default configuration that has this new template as well. This default configuration is the starting point for both our tuning method and random sampling. For $instances_B$ it is important that all vehicles are used and that the geographical division is reasonable logic from a planners perspective. The construction phase has been adopted to match these needs. This change does not significantly change (the size of) our search space. Moreover, the performance can be measured by a single value that has to be minimized.

In this experiment we again measure the indexed performance. An indexed performance of 100 represents the score of the default configuration and a lower indexed performance represents better performance. Random sampling results in an indexed performance of 85.59 and our tuning method results in an indexed performance of 85.58. In this experiment, our tuning method performs slightly better than random sampling, but the difference is not noteworthy. Hence, we conclude that the performance of our tuning method and random sampling are almost similar. That is, spending t time on building forests and generating promising configurations yields similar performance as spending t time on measuring (additional) configurations.

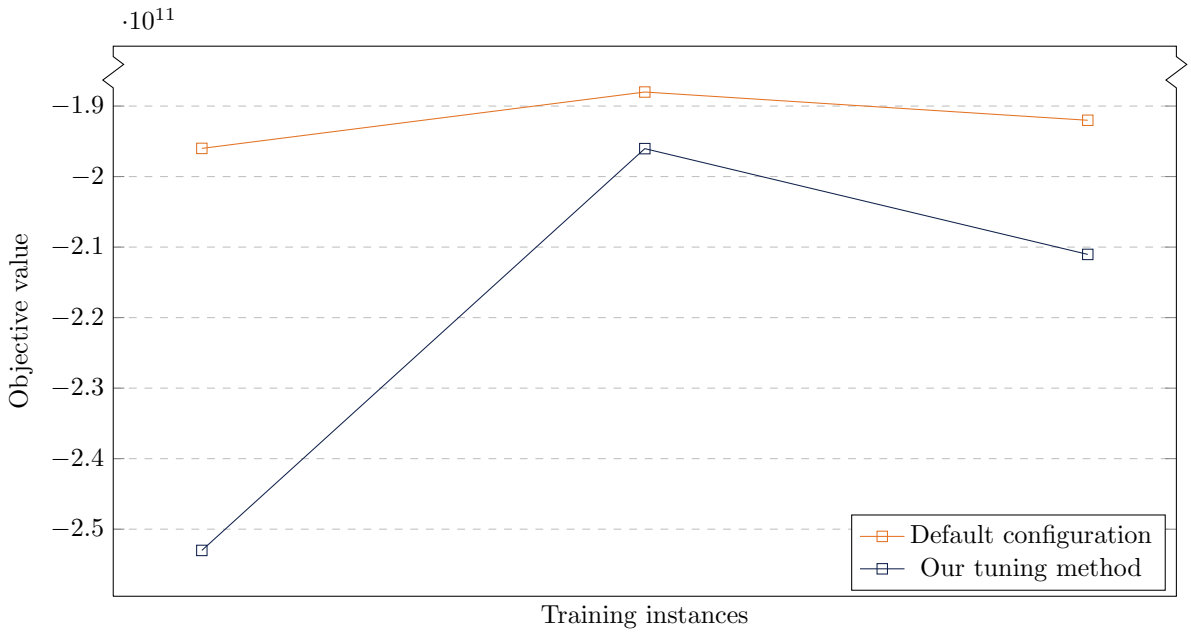

Figure 6.5: The performance on the 3 training instances for $instances_B$, for both the default configuration and our tuned configuration.

Robustness

Figure 6.5 shows the objective value for all 3 training instances, both with the default configuration and our tuned configuration. This figure clearly shows that our tuned configuration performs better on all training instances. Hence, we again conclude that our tuning method yields a configuration that performs robust.

6.3.3 Conclusion

Now that we conducted experiments with our tuning method for $instance_A$ and $instances_B$, we are able to conclude when our tuning method yields benefit over random sampling. In the experiment with $instance_A$, our tuning method resulted in a fairly large advantage over random sampling (see Figure 6.3). However, this advantage is negligible in the experiment with $instances_B$. The main difference between both experiments in the size of the training set: 25 instances and 3 instances respectively. This is also the main reason for the difference in performance between both. Our tuning method has an advanced mechanism, the intensification mechanism, to determine whether to perform more measurements with the current configuration (on other instances) or to generate new promising configurations (see Section 5.1.2). Random sampling randomly chooses on how many instances to measure a certain configuration. As the number of training instances grows, a good intensification mechanism becomes increasingly important. To illustrate this, consider the following example. Suppose we have a training set with 25 instances. Our tuning method would start with measuring a configuration on 1 instance only, and would increase the number of instances to measure on as long as the configuration looks promising. Random sampling, on the other hand, determines the number of instances to measure on regardless of the performance on the first instance(s) measured. Now suppose that random sampling chooses to measure a configuration on 15 instances, but that it is already evident after a few measurements that the current configuration performs very bad. In such cases, a lot of measurements are spent on this bad performing configuration. This problem does not occur if the size of the training set is small (e.g., 3 instances), since we cannot spend very much measurements on a bad performing configuration. For example, if the training set

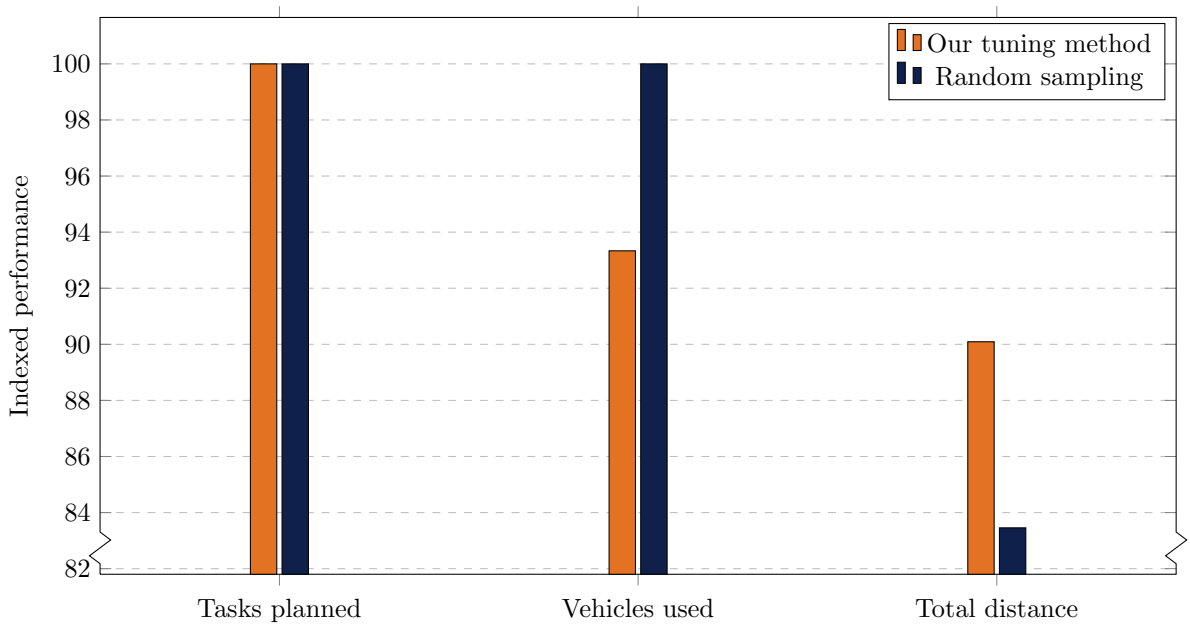

Figure 6.6: The indexed performance per tuning method for the various objectives on the test set of $instance_A$. An indexed performance of 100 represents the score of the default configuration. A lower indexed performance represents better performance.

consists of 3 instances we waste (i.e., spent unwise) at most 2 measurements, since 1 measurement is always performed. We therefore conclude that our tuning method makes better choices (i.e., parameter values) compared to random sampling, if the size of the training set is not too small.

6.4 Performance of our tuned configurations

In Section 6.3, we addressed the performance of our tuning method by comparing it with other tuning method. We compared the performance on the training instances. In this section, we assess the performance of our tuned configurations. That is, we examine how well the configurations found by our tuning method perform on the test instance(s) compared to other configurations. We use the same training and test sets as in Section 6.3.

6.4.1 $Instance_A$

For $instance_A$ we compare the configuration found by our tuning method with:

The default configuration The default configuration of CVRS, as introduced in Section 4.1.

The configuration found by random sampling The best configuration found by random sampling. We employ random sampling in the same way as in Section 6.3.

Figure 6.6 shows our results. Recall that an indexed performance of 100 represents the score of the default configuration. The performance on the first objective is is equal for all three cases. A detailed look at the results shows that even with the default configuration all tasks are planned. Hence, improvements are not possible on this objective (but deteriorations are). The benefit of our tuning method becomes clear if we consider the second objective: our tuned configuration yields an improvement of over 6% compared to both the default configuration and the configuration found by random sampling.

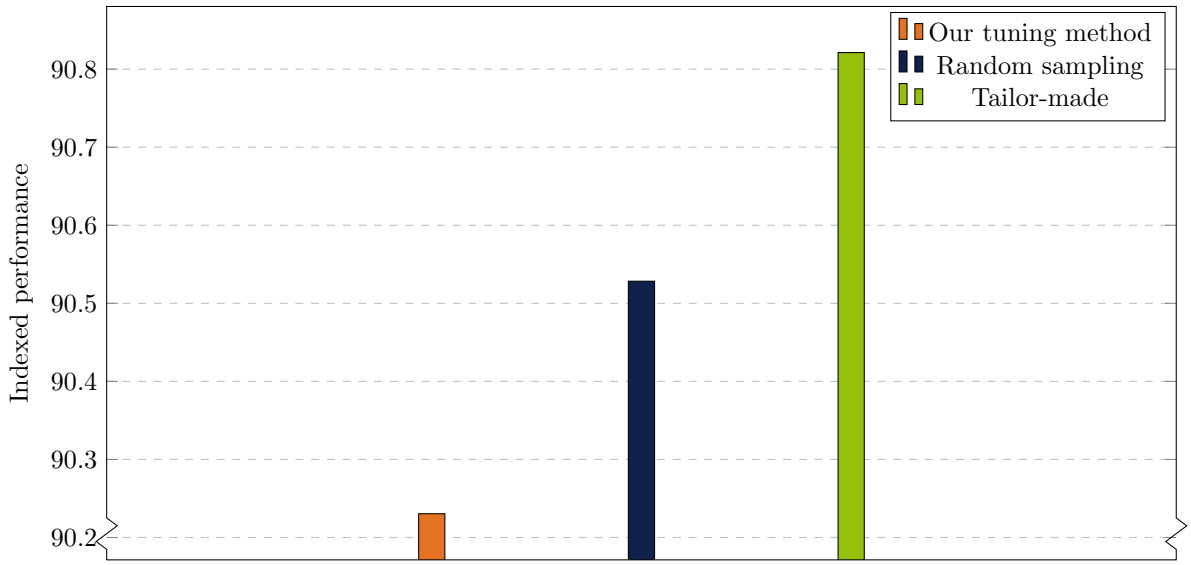

Figure 6.7: The indexed performance per tuning method for the various objectives on the test set of $instances_B$. An indexed performance of 100 represents the score of the default configuration. A lower indexed performance represents better performance.

6.4.2 $Instances_B$

For $instances_B$ we compare the configuration found by our tuning method with:

The default configuration The default configuration with the structure of the tailor-made configuration.

The configuration found by random sampling The best configuration found by random sampling. We employ random sampling in the same way as in Section 6.3, but use the structure of the tailor-made configuration.

The tailor-made configuration For these instance, ORTEC has tailor-made a configuration that performs very well for these specific instances.

Figure 6.7 shows our results. First of all, we notice that each tuning method is able to find a configuration that performs better than the default configuration. Our tuning method outperforms both random sampling and manual tuning (i.e., the tailor-made configuration). For $instances_B$, tuning yields an advantage of almost 10% compared to the default configuration.

The configuration found by our tuning method performs about 1.3% better than the tailor-made configuration. To estimate if much larger improvements are possible (i.e., if the optimal configuration performs much better), we look at the performance of the incumbent configuration with our tuning method over time (see Figure 6.8). Within two hours, our tuning methods finds a configuration that performs better than the tailor-made configuration. Moreover, the incumbent configuration that is found after about 20 hours, is the 119th configuration tried, and 158 measurements preceded it. This configurations remains the incumbent configuration for the last 40 hours as well. Hence, during these 40 hours the tuning method was not able to find a better performing configuration. During these 40 hours the predictions made by the random forests have a high likelihood of being reasonably good, since a lot of measurement data has been gathered already. As a consequence, the promising configurations are indeed good configurations, but they are not an improvement compared to the incumbent configuration. We therefore conclude that the incumbent configuration found by our tuning method is close to optimal.

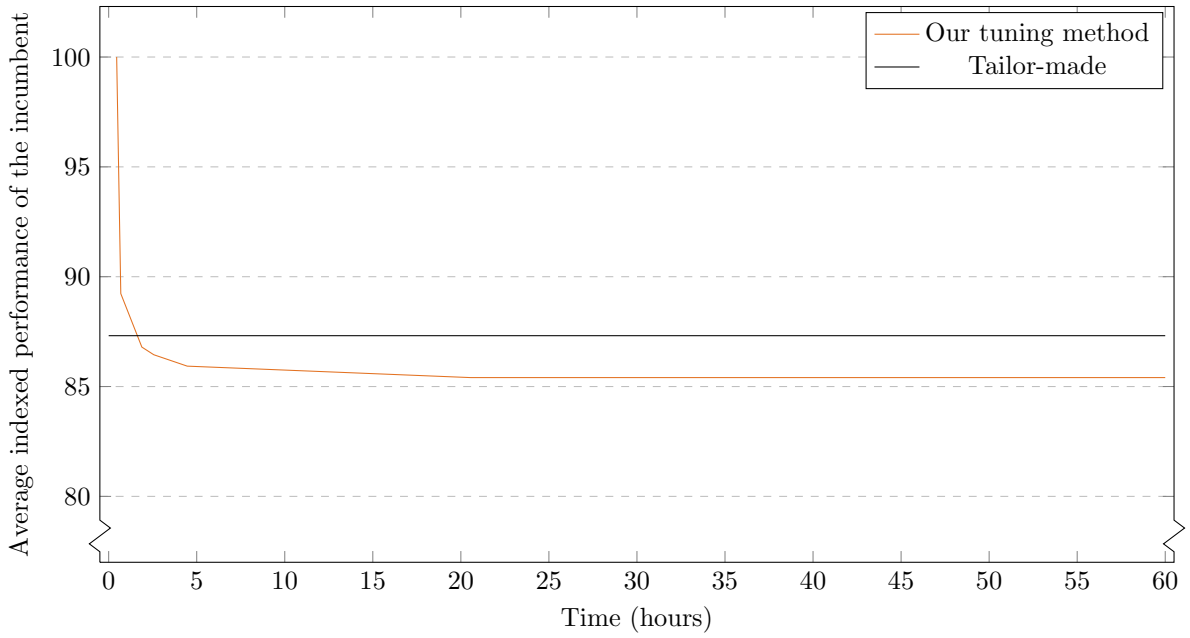

Figure 6.8: The average indexed performance of the incumbent configuration over time. An indexed performance of 100 represents the score of the default configuration. A lower indexed performance represents better performance.

6.4.3 Conclusion

For both $instance_A$ and $instances_B$, tuning always yielded an improvement over the default configuration. This indicates the need for tuning, since the performance of the default configuration can easily be improved. Moreover, it shows that using customizable configurations as opposed to a fixed, unchangeable algorithm is beneficial. Our tuning method outperformed the default configuration, the configurations found by random sampling, and the tailor-made configuration, in all conducted experiments. Using our tuning method is therefore the most efficient way to spend the 60 hours. Note that the tailor-made configuration was made in less than 60 hours, but since this was done by experts instead of just computational resources, this process was more expensive than our tuning method. Moreover, Figure 6.8 shows that we actually do not need the full 60 hours.

Based on Section 6.3.1 and Section 6.4.1, we again conclude that our training set is representative for our test instance in the case of $instance_A$: better performance on the training set results in better performance on test instance. The same conclusion holds for $instances_B$ (see Section 6.3.2 and Section 6.3.2). Based on these findings, we conclude that our tuning method is able to find configurations that perform well for the problem instances of a certain customer. Hence, tuning based on previous instances (e.g., those from the previous week) results in a configuration that performs well on future, unseen instances (e.g., this week) as well.

6.5 Versatility

Since ORTEC has many products, which all face similar tuning problems, we now test the versatility of SMAC by using it on other problem types, and domains. For these experiments we use the original version of SMAC, as our modifications were aimed at CVRS. The problems we selected are found in ongoing research at ORTEC. To test the method, we first turn to routing benchmark problems in Section 6.5.1. Next, in Section 6.5.2, we tune a vehicle clustering algorithm. In Appendix A, we present two other applications of SMAC for which we expect SMAC to become beneficial.

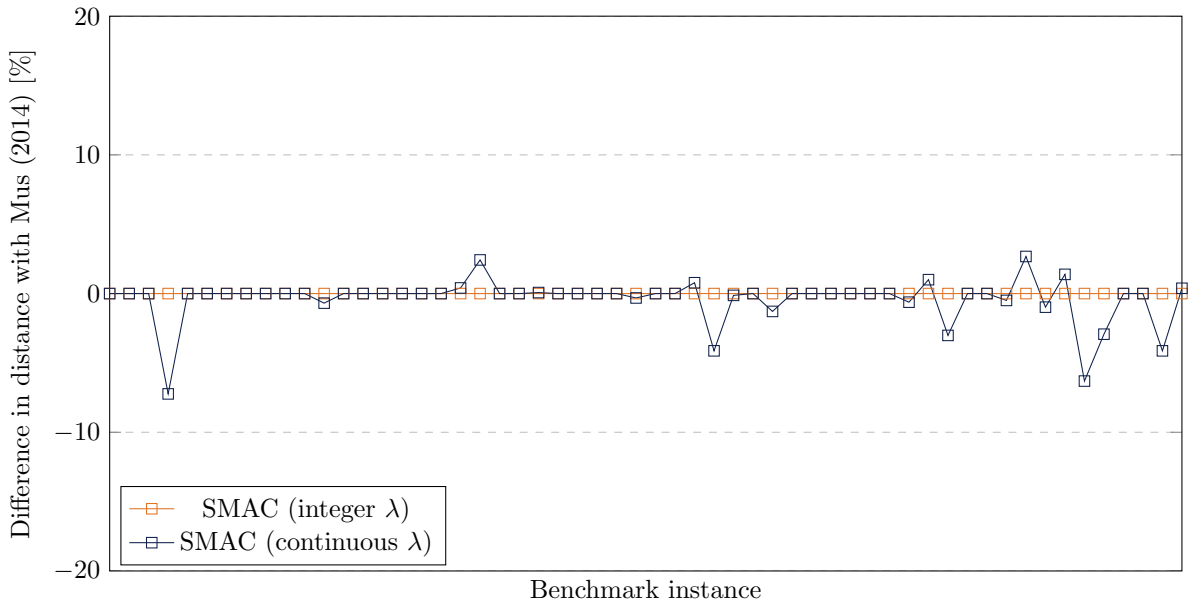

Figure 6.9: Comparison between Mus (2014) and SMAC.

6.5.1 Routing benchmark instances

We test our tuning method on the problem of Mus (2014): choosing the best configuration in a parameterized framework to solve the well-known Solomon (Solomon, 1987) benchmark instances for the VRPTW. Since the aim of Mus (2014) was not related to efficiently choosing parameters, complete enumeration was used to find the best configuration. In the framework, a choice should be made out of 24 possible configurations (hence, a categorical parameter) and 1 numerical parameter λ. To be able to use complete enumeration, the domain of λ was discretized (with a step size of 1), although this was not necessary for the framework. As a consequence, the best configuration found does not have to be the optimal configuration. Furthermore, in order to limit the time needed to find the best configuration, the problem was split into two sub-problems. First, the best configuration was found by omitting λ. Second, given the best configuration, the best value for λ was found. Therefore, the interaction effect between the configuration and λ was omitted.

To make a fair comparison, we have employed our tuning method in two ways. First, with a search space of 24 configurations and a *integer* λ on the domain [0,15]. Second, with the same configurations but with a λ on the continuous domain [0,15]. In both cases we limit the time allowed for tuning to 2.5 hours, which is less than 1% of the time needed to explore the full search space (using complete enumeration), with an integer λ. Where Mus (2014) split the problem into two sub-problems, we consider the whole search space at once, whereby we take possible interaction effects between a configuration and λ into account, which was ignored by Mus (2014).

Figure 6.9 shows our results. The figure shows the difference in distance with the configuration found by Mus (2014). For all benchmark instances it holds that a lower difference in distance represents better performance. From our results we notice that our tuning method with an integer λ yields the same results as the complete enumeration by Mus (2014): the difference in distance is 0 for all instances. Consequently, we conclude that our tuning method indeed finds the optimal configuration, given the search space with an integer λ. Moreover, if we allow a continuous domain for λ, our tuning method finds a configuration that is even better (the average difference is distance is lower than 0 in this case).

In this case, we showed that our tuning method significantly decreased the time needed to find the best configuration.

6.5.2 Customer clustering

The research of Besseling (2014) concerns a tactical routing problem, regarding network design. Given a set of customers with the frequencies they are visited, optimal delivery days should be calculated, such that the workload over time is streamlined and the total distance is minimized. For this purpose, Besseling (2014) developed an algorithm to cluster customers. This algorithm has, among others, two important parameters: one for the number of customers in a cluster and one for the number of neighbors to consider.

The algorithm may create infeasible solutions. To deal with this, we use a multi-objective approach. The first objective aims at making sure that the solution is feasible, the next objective is related to the workload, and the last objective concerns the total distance. We use the approach presented in Section 5.3.7 to take the multiple objectives into account.

To test the algorithm, a set of 5 uncorrelated instances is used. Since these instances are uncorrelated, we have to modify our approach slightly. We seek a set of parameters that performs well on all types of instances, so we use a training set consisting of all 5 instances. Since we tune on a variety of instances, we assume that we find a set of parameters that performs well overall, so that any future instances are also solved fairly.

Besseling (2014) optimizes the two parameters for each instance individually. These results show that the values for the parameters differ strongly among the instances. To find a set of parameters that creates a feasible solution for each instance, Besseling (2014) iterates over the possible parameter values, an expensive method. Although this set results in feasible solutions, the subsequent objectives (balancing the workload and minimizing the total distance) are not optimized. We solve the same problem using our tuning method. Our results show that we are able to find a set that creates a feasible solution for all instances and that performs better on the other objectives than the set found by Besseling (2014). To conclude, we are able to find a better set of parameters with less computation time.

In this case, we showed that our tuning method is able to find feasible, optimized parameter values, while taking multiple objectives into account.

Chapter 7

Conclusions and Recommendations

In Section 1.3.2 we outlined our objective: designing and implementing a method, able to automatically configure the routing algorithms to optimize their behavior on future instances. In this chapter we conclude to what extent we succeeded in achieving this objective. Moreover, we give recommendations to ORTEC and directions for further research to further develop our tuning method.

7.1 Conclusions

The objective was to design and implement a tuning method that performs well on future instances. During our research, we found out that a method was needed, able to handle categorical parameters and multiple instances. To develop a suitable method, we extended the work on SMAC (Hutter et al., 2011). SMAC has, to our knowledge, never been applied to the VRP or a framework comparable to CVRS. Using this technique in this are is therefore innovative, of which the results are unknown before.

Our tuning method is largely based on SMAC, but we made some modifications to improve the performance in our case. As Chapter 6 revealed, we indeed developed a method that matches our objective: our new intensification mechanism is beneficial in the case of CVRS. The results we achieved with our tuning method were very promising: we improved the configurations found by any other method for all conducted experiments, including the precursor of this research uRace (van Dijk, 2014). To gain the most benefit with our tuning method, the size of the training set should not be too small.

An important aspect of the tuned configurations is that they do not only perform well on the training instances, but also on unseen instances. Our experiments showed that using instances from only a few days leads to configurations that perform well on future instances. Hence, different instances of a certain customer all share a certain structure. Our tuning method is able to discover this structure and to find a suitable configuration for it. In addition, we showed that our design of the training and test set for $instance_A$ was valid. By this, ORTEC is able to tune a configuration based on limited customer data. This is especially useful in the implementation phase at a customer.

In addition, we showed that SMAC can also be use in combination with other ORTEC products (see Section 6.5). This has significantly improved the practical relevance of this research.

With this research we proved that SMAC, with some modifications, can successfully be employed in combination with a variety of practical frameworks. The tuning method outperformed any other tested method and can be employed significantly cheaper than the current tuning method (by experts). In addition, the configurations found with our tuning method perform better than those made by experts.

7.2 Recommendations

Since we already used data from two customers of ORTEC for our experiments, the experimental results are valid in practice. Since we showed that the configurations are tuned best with our tuning method, we recommend ORTEC to implement this method. Customers benefit from this solution, since their instances are solved best with a configuration found by our tuning method. This in turn may lead to cost reductions at the customer side. Moreover, our tuning method leads to cost reductions for ORTEC as it reduces the expensive time of experts needed for tuning configurations.

In Section 7.2.1, we outline how ORTEC should implement our tuning method. Section 7.2.1 discusses the frequency with which our tuning method should be used. In Section 7.2.2, we present details that ORTEC should take into account when using our tuning method on other domains.

7.2.1 Usage scenarios

Currently, ORTEC's consultants and developers (from now on, we refer to them as experts) make the tailor-made configurations. Often they start with the default configuration and tune this to make it suitable for a customer. We do not want to fully replace those consultants and developers, as we assume that their knowledge is valuable for tuning other configurations. Hence, we propose to implement our tuning method to support them. We distinguish two moments, called usage scenarios, in the tuning process where it is appropriate to use our tuning method: (i) before tuning by the experts or (ii) after tuning by the experts. We outline these scenarios in this section.

Before experts

Currently, the experts use the default configuration as their starting point. Using our tuning method, we can improve their starting point by running the tuning method with the default configuration as a starting point. The configuration found after tuning can be used by the experts as their starting point. This configuration already performs better for the corresponding customer than the default configuration. If experts start with this improved configuration, they are able to recognize certain patterns in this configuration. It is likely that experts can create an even better configuration, by combining those patterns with the knowledge they obtained with tuning in previous cases. Summarized, this usage scenario gives experts better insight into which patterns are suitable for a certain customer, before they start tuning themselves.

After experts

Once our tuning method starts with a certain configuration, it is almost impossible that it ends up with a configuration that performs worse than the starting configuration. Hence, if we use our tuning with a configuration made by experts as a starting point, we might be able to improve this configuration. By using the experts knowledge first, it is expected that our tuning method learns more quickly which combination of parameters performs well. As a consequence, it will measure more good performing configurations. Summarized, this usage scenario gives our tuning method a head start, such that less poor performing configurations are measured.

Conclusion

The choice for a usage scenario strongly depends on the context. For example, if it is hard to find improvements over the default configuration, the first usage scenario is advisable. Otherwise, the second usage scenario is advisable as it gives our tuning method a head start. The two usage scenarios we

discussed both have their advantages and are not mutually exclusive: they can be used together. To revise the tailor-made configurations currently used by customers, opting for the second usage scenario is evident.

Frequency

Tuning configurations is a time (and hence money) consuming task in the current situation. Therefore, they are not often revised. Our tuning method only requires computational resources. If an abundance of computational resources is available, using our tuning method is cheap. Revising configurations can therefore be done more often with our tuning method. Changes in the problem instances of customers, of which the customer might not even be aware of, can be recognized and the configuration can be adapted. It is therefore advisable to tune during all weekends, since an abundance of computational resources is generally available during the weekend. After the weekend, ORTEC's customers start with a completely optimized configuration, ready to solve the problems of the upcoming week.

In our experiments we used a full weekend for tuning. As Figure 6.8 shows, this is not necessary. Hence, customers should be free to choose the time they want to invest in tuning, while being aware of the fact that too little time does not take full advantage of the tuning method. To increase the performance of our tuning method when little time is available, a so-called warmstart can be used. That is, instead of starting with an empty forest (i.e, no measurements data), we can start with the measurements data gathered in previous runs of our tuning method. This will speed up our tuning method, since we can make better choices from the beginning. When using this, one should be aware that extremely old instances are removed from the training set as they might not be representative anymore.

7.2.2 Other domains

We showed that our tuning method not only performs well for CVRS, but that it can also be used on other domains. Nevertheless, we recommend to carefully analyze SMAC when using it on other domains, as modifications might be possible that increase the performance of our tuning method for a specific domain. In fact, this is what we did with SMAC: we adapted it make it suitable for CVRS. We showed that these modifications lead to performance improvements. Hence, we assume that similar improvements can be made on other domains, we choosing relevant modifications for SMAC.

ORTEC also has customers for which solving an instance takes more than one hour (hence, the computation time limit does not count here). For such cases, tuning is more difficult: in the same amount of time less performance measurements can be made, so the performance of the found configuration will not be as good as for cases with a smaller computation time. For such cases, we recommend to use a parallel implementation (Hutter, Hoos, & Leyton-Brown, 2012) of our tuning method: executing multiple performance measurements at the same time. Research has show that this significantly decreases the time needed to find competitive configurations.

7.3 Further research

During our research we excluded many possibilities to make our tuning method even better. In this section, we discuss some of these possibilities.

SMAC has many options to influence its working. For example, one could vary the number of trees in the forest, and the size of the subset of parameters to take for each tree. During our research, we left all these options at their default values. These default values have proven to work well in many cases by other researchers. Nevertheless, the fact remains that other values may improve the working of the

method in our cases. Research aiming at optimizing these parameters for specific cases may lead to an increased performance of our tuning method.

In Section 5.3.4 we discussed our local search procedure to generate promising configurations. Our search space consisted of many categorical parameters. The effect of a certain categorical parameter value cannot be estimated based on other measured values for that parameter: the effect of a parameter values should be measured in order to be able to draw conclusions about it. However, numerical parameters generally allow for interpolation and extrapolation techniques, making those parameters less expensive to explore. Advanced mathematical local search techniques, which generally perform better in search spaces with only numerical parameters, could therefore not be used in our research. Nevertheless, ORTEC may have tuning problems that consist of solely numerical parameters. The use of advanced mathematical local search techniques could increase the performance of our tuning method for such problems.

In Section 7.2.1 we proposed to use a warmstart to speed up our tuning method. Although it is obvious that this does indeed speed up the tuning process, the extent to which is difficult to estimate. In order to find a good balance between the invested tuning time and the performance, additional research is needed. Results of this research may even lead to using our tuning method as an online tuning method, if (i) a warmstart significantly reduces the tuning time needed for good performance and (ii) customer are willing to wait a little longer for an improved solution.

References

Altinel, I., & Öncan, T. (2005). A new enhancement of the Clarke and Wright savings heuristic for the capacitated vehicle routing problem. *Journal of the Operational Research Society*, *56*(8), 954-961.

Ansótegui, C., Sellmann, M., & Tierney, K. (2009). A gender-based genetic algorithm for the automatic configuration of algorithms. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *5732 LNCS*, 142-157.

Baker, B., & Ayechew, M. (2003). A genetic algorithm for the vehicle routing problem. *Computers and Operations Research*, *30*(5), 787-800.

Baker, B., & Sheasby, J. (1999). Extensions to the generalised assignment heuristic for vehicle routing. *European Journal of Operational Research*, *119*(1), 147-157.

Balaprakash, P., Birattari, M., & Stützle, T. (2007). Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *4771 LNCS*, 108-122.

Baldacci, R., Christofides, N., & Mingozzi, A. (2008). An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts. *Mathematical Programming*, *115*(2), 351-385.

Baldacci, R., Hadjiconstantinou, E., & 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.

Balinski, M., & Quandt, R. (1964). On an Integer Program for a Delivery Problem. *Operations Research*, *12*(2), 300-304.

Bartz-Beielstein, T., Lasarczyk, C., & Preuss, M. (2005). Sequential parameter optimization. In (Vol. 1, p. 773-780).

Bartz-Beielstein, T., & Markon, S. (2004). Tuning search algorithms for real-world applications: A regression tree based approach. In *Proceedings of CEC-04* (p. 1111-1118).

Bent, R., & Van Hentenryck, P. (2004). A two-stage hybrid local search for the vehicle routing problem with time windows. *Transportation Science*, *38*(4), 515-530.

Besseling, A. (2014). *Tactical Routing, a way to improve* (Master's thesis). University Amsterdam.

Birattari, M., Stützle, T., Paquete, L., & Varrentrapp, K. (2002). A Racing Algorithm for Configuring Metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference* (p. 11-18). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Bramel, J., & Simchi-Levi, D. (1995). A Location based Heuristic for General Routing Problems. *Operations Research*, *43*(4), 649-660.

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5-32.

Bullnheimer, B., Hartl, R., & Strauss, C. (1999). An improved Ant System algorithm for the Vehicle

Routing Problem. *Annals of Operations Research*, *89*, 319-328.

Burke, E., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., & Qu, R. (2013). Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, *64*(12), 1695-1724.

Burke, E., Hyde, M., & Kendall, G. (2012). Grammatical evolution of local search heuristics. *IEEE Transactions on Evolutionary Computation*, *16*(3), 406-417.

Burke, E., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., & Woodward, J. (2010). A Classification of Hyper-heuristic Approaches. In M. Gendreau & J.-Y. Potvin (Eds.), *Handbook of Metaheuristics* (Vol. 146, p. 449-468). Springer US.

Buttrey, S. (1998). Nearest-neighbor classification with categorical variables. *Computational Statistics and Data Analysis*, *28*(2), 157-169.

Christofides, N. (1976). VEHICLE ROUTING PROBLEM. *Rev Fr Autom Inf Rech Oper*, *10*(2), 55-70.

Christofides, N., & Eilon, S. (1969). An Algorithm for the Vehicle-Dispatching Problem. *OR*, *20*(3), 309-318.

Christofides, N., Mingozzi, A., & Toth, P. (1981). Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations. *Mathematical Programming*, *20*(1), 255-282.

Clarke, G., & Wright, J. (1964). Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. *Operations Research*, *12*(4), 568-581.

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

Cordeau, J.-F., Laporte, G., & Mercier, A. (2001). A unified tabu search heuristic for vehicle routing problems with time windows. *Journal of the Operational Research Society*, *52*(8), 928-936.

Cowling, P., Kendall, G., & Soubeiga, E. (2001). A hyperheuristic approach to scheduling a sales summit. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *2079 LNCS*, 176-190.

Croes, G. (1958). A Method for Solving Traveling-Salesman Problems. *Operations Research*, *6*(6), 791-812.

Dantzig, G. B., & Ramser, J. H. (1959). The Truck Dispatching Problem. *Management Science*, *6*(1), 80-91.

van Dijk, T. (2014). *Tuning the Parameters of a Loading Algorithm* (Master's thesis). University of Twente.

Dobslaw, F. (2010). Recent Development in Automatic Parameter Tuning for Metaheuristics. In *Proceedings of the 19th Annual Conference of Doctoral Students - WDS 2010* (p. 54-63).

Drake, J., Kililis, N., & Özcan, E. (2013). Generation of VNS components with grammatical evolution for vehicle routing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *7831 LNCS*, 25-36.

Eiben, A., & Smit, S. (2012). Evolutionary Algorithm Parameters and Methods to Tune Them. In Y. Hamadi, E. Monfroy, & F. Saubion (Eds.), *Autonomous Search* (p. 15-36). Springer Berlin Heidelberg.

Emmen, R. (2014). *Univariate Forecasting of Sales Volumes for Personnel Planning* (Master's thesis). University Amsterdam.

Ergun, Ö., Orlin, J., & Steele-Feldman, A. (2006). Creating very large scale neighborhoods out of smaller ones by compounding moves. *Journal of Heuristics*, *12*(1-2), 115-140.

Fisher, M. (1994). Optimal solution of vehicle routing problems using minimum K-trees. *Operations Research*, *42*(4), 626-642.

Fisher, M., & Jaikumar, R. (1978). *A Decomposition Algorithm for Large-scale Vehicle Routing.* Department of Decision Sciences, Wharton School, University of Pennsylvania.

Fisher, M., & Jaikumar, R. (1981). GENERALIZED ASSIGNMENT HEURISTIC FOR VEHICLE ROUTING. *Networks*, *11*(2), 109-124.

Frazier, P., Powell, W., & Dayanik, S. (2008). A knowledge-gradient policy for sequential information collection. *SIAM Journal on Control and Optimization*, *47*(5), 2410-2439.

Fukasawa, R., Longo, H., Lysgaard, J., De Aragão, M., Reis, M., Uchoa, E., & Werneck, R. (2006). Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical Programming*, *106*(3), 491-511.

Gambardella, L. M., Taillard, É., & Agazzi, G. (1999). MACS-VRPTW: A Multiple Colony System For Vehicle Routing Problems With Time Windows. In *New Ideas in Optimization* (p. 63-76). McGraw-Hill.

Garrido, P., & Castro, C. (2009). Stable solving of CVRPs using hyperheuristics. In (p. 255-262).

Garrido, P., Castro, C., & Monfroy, E. (2009). Towards a flexible and adaptable hyperheuristic approach for VRPs. In (Vol. 1, p. 311-317).

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

Gaskell, T. (1967). Bases for Vehicle Fleet Scheduling. *OR*, *18*(3), 281-295.

Gendreau, M., Hertz, A., & Laporte, G. (1992). New Insertion and Postoptimization Procedures for the Traveling Salesman Problem. *Operations Research*, *40*(6), 1086-1094.

Gendreau, M., Hertz, A., & Laporte, G. (1994). Tabu search heuristic for the vehicle routing problem. *Management Science*, *40*(10), 1276-1290.

Ghaziri, H. (2004). Solving routing problem by self-organizing maps. In T. Kohonen, K. Makisara, O. Simula, & J. Kangas (Eds.), *Artificial neural networks* (p. 829-834). Amsterdam, Netherlands: Elsevier.

Gillett, B., & Miller, L. (1974). A Heuristic Algorithm for the Vehicle-Dispatch Problem. *Operations Research*, *22*(2), 340-349.

Glover, F. (1992). New Ejection Chain and Alternating Path Methods for Traveling Salesman Problems. *Computer Science and Operations Research*, 449-509.

Gómez-Alonso, C., & Valls, A. (2008). A similarity measure for sequences of categorical data based on the ordering of common elements. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *5285 LNAI*, 134-145.

Gromicho, J., van Hoorn, J. J., Kok, A. L., & Schutten, J. M. J. (2012). Restricted Dynamic Programming: A Flexible Framework for Solving Realistic VRPs. *Computers & Operations Research*, *39*(5), 902-909.

Hamming, R. (1950). Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, *29*(2), 147-160.

Hansen, N., & Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, *9*(2), 159-195.

Held, M., & Karp, R. (1961). A Dynamic Programming Approach to Sequencing Problems. In *Proceedings of the 1961 16th ACM National Meeting* (p. 71.201-71.204). New York, NY, USA: ACM.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI, USA: University of Michigan Press.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *6683 LNCS*, 507-523.

Hutter, F., Hoos, H., Leyton-Brown, K., & Murphy, K. (2009). An Experimental Investigation of Model-based Parameter Optimisation: SPO and Beyond. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation* (p. 271-278). New York, NY, USA: ACM.

Hutter, F., Hoos, H., Leyton-Brown, K., & Stützle, T. (2009). ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, *36*, 267-306.

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2012). Parallel Algorithm Configuration. In *Proceedings of the 6th International Conference on Learning and Intelligent Optimization* (p. 55-70). Springer-Verlag.

Jones, D., Schonlau, M., & Welch, W. (1998). Efficient Global Optimization of Expensive Black-Box Functions. *Journal of Global Optimization*, *13*(4), 455-492.

Kok, A., Meyer, C., Kopfer, H., & Schutten, J. (2010). A Dynamic Programming Heuristic for the Vehicle Routing Problem with Time Windows and European Community Social Legislation. *Transportation Science*, *44*(4), 442-454.

Kytöjoki, J., Nuortio, T., Bräysy, O., & Gendreau, M. (2007). An efficient variable neighborhood search heuristic for very large scale vehicle routing problems. *Computers and Operations Research*, *34*(9), 2743-2757.

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. (2007). What you should know about the vehicle routing problem. *Naval Research Logistics*, *54*(8), 811-819.

Laporte, G. (2009). Fifty years of vehicle routing. *Transportation Science*, *43*(4), 408-416.

Laporte, G., & Nobert, Y. (1987). Exact Algorithms for the Vehicle Routing Problem. *North-Holland Mathematics Studies*, *132*(C), 147-184.

Laporte, G., Nobert, Y., & Desrochers, M. (1985). OPTIMAL ROUTING UNDER CAPACITY AND DISTANCE RESTRICTIONS. *Operations Research*, *33*(5), 1050-1073.

Leyton-Brown, K., Nudelman, E., & Shoham, Y. (2009). Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, *56*(4).

Likert, R. (1932). A Technique for the Measurement of Attitudes. *Archives of Psychology*, *22*(140), 1-55.

Meignan, D., Koukam, A., & Créput, J.-C. (2010). Coalition-based metaheuristic: A self-adaptive metaheuristic using reinforcement learning and mimetism. *Journal of Heuristics*, *16*(6), 859-879.

Mes, M., Powell, W., & Frazier, P. (2011). Hierarchical Knowledge Gradient for Sequential Sampling. *Journal of Machine Learning Research*, *12*, 2931-2974.

Mester, D., & Bräysy, O. (2007). Active-guided evolution strategies for large-scale capacitated vehicle routing problems. *Computers and Operations Research*, *34*(10), 2964-2975.

Mlejnek, J., & Kubalík, J. (2013). Evolutionary hyperheuristic for capacitated vehicle routing problem. In (p. 219-220).

Mockus, J. (1994). Application of Bayesian approach to numerical methods of global and stochastic optimization. *Journal of Global Optimization*, *4*(4), 347-365.

Mus, M. (2014). *Vehicle Routing Problem With Time Windows: Escape local minima* (Master's thesis). University Amsterdam.

Nannen, V., & Eiben, A. (2006). A Method for Parameter Calibration and Relevance Estimation in Evolutionary Algorithms. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation* (p. 183-190). New York, NY, USA: ACM.

O'Neill, M., & Ryan, C. (2001). Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, *5*(4), 349-358.

Or, I. (1976). *Traveling Salesman-Type Combinatorial Problems and their Relation to the Logistics of Regional Blood.* (Ph.D. Thesis)

Osman, I. (1993). Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, *41*(4), 421-451.

Pisinger, D., & Ropke, S. (2007). A general heuristic for vehicle routing problems. *Computers and Operations Research*, *34*(8), 2403-2435.

Potvin, J.-Y., & Robillard, C. (1995). Clustering for vehicle routing with a competitive neural network. *Neurocomputing*, *8*(2), 125-139.

Rochat, Y., & Taillard, E. (1995). Probabilistic diversification and intensification in local search for vehicle routing. *Journal of Heuristics*, *1*(1), 147-167.

Sabar, N., Ayob, M., Kendall, G., & Qu, R. (2013). Grammatical evolution hyper-heuristic for combinatorial optimization problems. *IEEE Transactions on Evolutionary Computation*, *17*(6), 840-861.

Schumann, M., & Retzko, R. (1995). Self-organizing maps for vehicle routing problems – minimizing an explicit cost function. In *Proceedings of the International Conference on Artificial Neural Networks* (p. 401-406). Paris, France.

Smit, S., & Eiben, A. (2009). Comparing parameter tuning methods for evolutionary algorithms. In (p. 399-406).

Solomon, M. (1987). Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints. *Operations Research*, *35*(2), 254-265.

Styles, J., & Hoos, H. (2013). Using racing to automatically configure algorithms for scaling performance. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *7997 LNCS*, 382-388.

Taillard, É. (1993). Parallel iterative search methods for vehicle routing problems. *Networks*, *23*(8), 661-673.

Taillard, É., Badeau, P., Gendreau, M., Guertin, F., & Potvin, J.-Y. (1997). A tabu search heuristic for the vehicle routing problem with soft time windows. *Transportation Science*, *31*(2), 170-186.

Thompson, P. M., & Psaraftis, H. N. (1993). Cyclic transfer algorithms for multivehicle routing and scheduling problems. *Operations Research*, *41*(5), 935-946.

Toth, P., & Vigo, D. (2003). The granular tabu search and its application to the vehicle-routing problem. *INFORMS Journal on Computing*, *15*(4), 333-346.

Vidal, T., Crainic, T., Gendreau, M., & Prins, C. (2014). A unified solution framework for multi-attribute vehicle routing problems. *European Journal of Operational Research*, *234*(3), 658-673.

Yellow, P. (1970). A Computational Modification to the Savings Method of Vehicle Scheduling. *Operational Research Quaterly*, *21*(2), 281-283.

Appendix A

Other applications

In this appendix we present two other applications of SMAC for which we expect SMAC to become beneficial. In Section A.1, we describe how SMAC can be applied to a forecasting engine. In Section A.2, we discuss how SMAC can be applied to an optimizer that solves instances that are not correlated with training instances.

A.1 Forecasting engine

The research of Emmen (2014) aims at designing a fully automated forecasting engine, such that it can be used by employees without forecasting knowledge. This engine can be used in combination with several applications of ORTEC. Each application requires a different forecasting method. Therefore, the internal design of the engine strongly depends on the application. Moreover, the problem instance on hand also influences the optimal design (like in our other experiments). Summarized, designing the forecasting engine for a specific application or instance is not obvious.

Within this engine, many parameters exist that influence the quality of the forecast. For example, a categorical parameter describing the main forecasting method to use, and numerical parameters regarding seasonality and trends. Emmen (2014) uses SMAC to find the optimal parameter value for a given, single instance. The main difficulty was that the choice of the method, the categorical parameter, influences the other parameters a lot. SMAC was able to find parameter values that result in a reliable forecast for multiple instances. However, the number of options for the categorical parameters was limited in the conducted experiments. Hence, optimizing all possible forecasting methods with a simple numerical tuning method would have yielded the same outcome. The use of an advanced method such as SMAC was therefore not necessary in this case. Nevertheless, we expect SMAC to become beneficial if the number of possible forecasting methods increases.

A.2 Cloud optimizer

In most previous cases we assumed that future, unseen instances were correlated with the training instances. We exploited this to find a promising configuration for the future, unseen instances.

ORTEC also has a vehicle optimizer in the cloud. This service is used by multiple customers, and there are several configurations that can be used. Therefore, future, unseen instances do not have a correlation with the previously seen instances, since they may be of different customers. Even instances for the same customer might not be correlated, since many business units can use the service (whose problems are totally different). Nevertheless, we still want to use the best configuration for this new

instance.

Currently, the configuration that should be used to solve the problem instance is part of the request to the service. Customers have to choose which configuration to choose for which instance, and this is difficult to determine. Moreover, the chosen configuration might not be the best available configuration for the problem instance. We can use a part of our tuning method to improve this process.

With our random forests, we are able to quickly predict the performance of a certain configuration on a certain problem instance, with the inclusion of instance features. Instance features are characteristics of an instance (e.g., number of tasks). This information is used when predicting the performance: the performance on similar instances is used instead of the performance on all instances.

In Section 5.1.1, we described our random forests procedure. For our new application, we have to slightly modify this procedure in the following way. Originally, we branched on the parameters of a configuration, since we wanted to estimate the performance of a configuration. In this new application, we want to estimate the performance of a configuration on a particular instance. We should therefore branch on characteristics of the instance as well. That is, we also take a random subset of the instance features to be eligible to be split upon. Also, the number of possible configurations is now limited and fixed: we branch on configurations instead of the individual configuration parameters. In this new situation we branch on both configurations and instance features, so that the performance of a configuration on a particular instance can be estimated.

We can use this model to let the optimizer decide which configuration to use, instead of the customer. That is, whenever a request comes in, we quickly predict the performance of all available configurations using our random forest model. The configuration that is expected to perform best is selected to solve the instance. In this case, the configuration selection is done online, i.e., while solving the problem. As a consequence, the solving time for an instance increases. However, since our random forest model is fast, we expect that the increased solution quality outweighs the additional time needed.

Unfortunately, we were not yet able to validate the results during our research. Nevertheless, the above conceptual model forms a good start for research in this field.