A Survey of Methods for Automated Algorithm Configuration (Extended Abstract) *

Elias Schede1, Jasmin Brandt2, Alexander Tornede3, Marcel Wever4,5, Viktor Bengs4,5, Eyke Hüllermeier4,5 and Kevin Tierney1

1Decision and Operation Technologies Group, Bielefeld University, Bielefeld, Germany
2Department of Computer Science, Paderborn University, Paderborn, Germany
3Institute of Artificial Intelligence, Leibniz University Hannover, Hannover, Germany
4Institute of Informatics, LMU Munich, Munich, Germany
5Munich Center for Machine Learning, Munich, Germany
{elias.schede, kevin.tierney}@uni-bielefeld.de, jasmin.brandt@upb.de, a.tornede@ai.uni-hannover.de, {marcel.wever, viktor.bengs}@ifi.lmu.de, eyke@lmu.de

Abstract

Algorithm configuration (AC) is concerned with the automated search of the most suitable parameter configuration of a parametrized algorithm. There are currently a wide variety of AC problem variants and methods proposed in the literature. Existing reviews do not take into account all derivatives of the AC problem, nor do they offer a complete classification scheme. To this end, we introduce taxonomies to describe the AC problem and features of configuration methods, respectively. Existing AC literature is classified and characterized by the provided taxonomies.

1 Introduction

In many industries and academic fields, difficult computational problems need to be solved on a regular basis. Examples of these problems include constraint satisfaction problems, Boolean satisfiability problems (SAT), vehicle routing problems, finding appropriate machine learning models for given datasets, and computing highly complex simulations. Algorithms developed to solve such problems have parameters that significantly affect the algorithm’s required runtime to solve problem instances or the quality of returned solutions. To achieve optimal results with respect to run time or solution quality, different sets of problem instances require different parameter values, also referred to as configurations. It is therefore crucial to adjust the algorithm’s configuration to the specifics of the problem instances at hand. This adjustment, however, is as complex task because the algorithm to be configured must be actually executed for different configurations to observe the target metrics (e.g., runtime or solution quality).

The research field of algorithm configuration (AC) has emerged to address the challenge of determining suitable parameter values for algorithms. Especially over the last two decades, many approaches and problem variants have been proposed in this field. Broadly speaking, AC approaches aim to efficiently find a good configuration for an algorithm to recommend that configuration for new, unseen problem instances at a later stage. To find a good configuration, typically a training set of problem instances is used in an offline phase. The training set can be used as input to run the algorithm with different configuration settings and observe the respective performance. It is hoped that these observations on the train set can be generalized to make good recommendations for production settings.

To illustrate the benefits of searching for algorithm parameters in an automated way, consider the circuit satisfiability problem as an example. This is a classic SAT problem, where the task is to find a value assignment such that the output of a Boolean circuit evaluates to true [Marques-Silva, 2008]. In a business application, many such circuits must often be evaluated for feasibility in limited time. To do this, an efficient SAT solver such as Glucose [Audemard and Simon, 2009] is needed to provide solutions in a timely manner. Glucose in turn exposes several parameters that influence the search for assignments and, when set correctly, can speed up the search significantly.

Indeed, configurations for SAT solvers found by ParamILS [Hutter et al., 2007b], one of the first procedures to search for high-quality parameters in a structured way, lead to considerable speedups compared to default configurations of solvers. In particular, the configurations found reduce the arithmetic mean runtime for software verification instances for the SAT solver SPEAR from 787.1s to 1.5 seconds in the best case [Hutter et al., 2007a]. More recently, PyDGGA [Ansotegui et al., 2021] reduced the solving time of the SAT solver SparrowToRiss [Balint and Manthey, 2013] on instances from the N-Rooks [Lindauer and Hutter, 2018] dataset from 116 to 6.3 seconds. AC thus offers a simple way of squeezing extra performance out of existing algorithms for specific datasets.

---

2 Problem Description

2.1 AC

To define the AC problem more formally, we introduce the following notation that is similar to [Hutter et al., 2009]. Let \( I \) be a space of problem instances over which a probability distribution \( P \) is defined. Optional feature vectors \( f_i \in \mathbb{R}^d \) with features \( f_{i,1}, ..., f_{i,d} \) can be computed for problem instances \( i \in I \) coming from this space. Furthermore, let \( A \) denote a parametrized target algorithm, with parameters \( p_1, ..., p_k \) which may be of categorical or numerical nature. The (finite or infinite) domain of each parameter \( p_i \) is denoted by \( \Theta_i \) such that \( \Theta_i \subseteq \Theta_1 \times ... \times \Theta_k \) is the space of all feasible parameter combinations, i.e., the so-called configuration or search space. A concrete instantiation of the target algorithm \( A \) with a given configuration \( \theta \in \Theta \) is denoted by \( A_\theta \). Furthermore, let \( c : I \times \Theta \rightarrow \mathbb{R} \) be a cost function from the space of cost functions \( C \), which quantifies the cost of running a given problem instance with a given configuration. Depending on the target algorithm, \( c \) may be stochastic and contain noise. Then, ideally, we would like to find the optimal configuration \( \theta^* \in \Theta \) defined as

\[
\theta^* = \arg \min_{\theta \in \Theta} \int I c(i, \theta) dP(i) .
\]  

(1)

However, in practice, the distribution \( P \) over \( I \) is unknown, and thus we must resort to solving a proxy problem. The aggregation function is usually the arithmetic mean or a variation thereof that is computed over the given problem instances by applying the given configuration to each of them and computing their cost. Similar to empirical risk minimization in machine learning, we then seek to find the configuration minimizing the aggregated costs across the training instances, i.e.,

\[
\hat{\theta} \in \arg \min_{\theta \in \Theta} m(c, I_{train}, \theta) .
\]  

(2)

Informally, the problem can be expressed as: given a target algorithm with a set of parameters and a set of problem instances, find a configuration that yields good performance with respect to the cost measure across the set of problem instances. We will refer to automated approaches capable of finding such configurations as (algorithm) configurators.

2.2 Scope

We select and review stand-alone AC methods that are suited to solve the problem described before. The identified methods and their features are examined and used to derive the classification scheme. We omit articles related to algorithm selection (AS) and hyperparameter optimization (HPO) since we consider these to be sub problems of AC for which comprehensive literature is available. An overview of AC and the relation between AS, HPO and CASH is given in Figure 1.

Algorithm Selection (AS) AS is a sub problem of AC, however, which we exclude here, since it has been considered in several reviews already [Kerschke et al., 2019; Kotthoff, 2016]. In fact, AS is special case of instance-specific AC, with a search space consisting of only one categorical parameter that represents the target algorithm choice.

In other words, AS aims at learning an algorithm choice that is tailored to the input instance, which limits its scope compared to AC. In particular, the search space in AS is typically small, discrete and consists of a (static) set of algorithms (although new extensions exist that handle larger spaces [Tornede et al., 2020]). On the other hand the search space in AC, is generally based on the parameters of one target algorithm, and thus, algorithm configurators need to be able to handle much larger, if not even infinite, search spaces [Kerschke et al., 2019].

Hyperparameter Optimization (HPO) We will not be discussing Hyperparameter Optimization (HPO) techniques in addition to Algorithm Selection (AS) because they have already been reviewed extensively in several papers [Yu and Zhu, 2020; Luo, 2016; Yang and Shami, 2020; Bischl et al., 2021]. Before defining HPO more clearly, let us have a closer look at the terminology around the words hyperparameter and parameter. In HPO, parameters that should be set by a user are called hyperparameters, while in the realm of AC these are typically referred to as parameters. In HPO the term hyperparameters is used since machine learning models usually also contain parameters that are induced from data and are not subject to configuration. In fact, it is this difference in terminology that leads us to one of the key differences between HPO and general AC, namely that AC methods focus on configuring target algorithms that solve instances of a dataset independently, while HPO learns hyperparameters for target algorithms that train parameters on multiple instances of a single dataset in tandem.

As HPO is a subset of the AC setting, HPO techniques can, in theory, be used to search for configurations in the general AC setting. However, in practice, this is rarely done because HPO methods lack two important functionalities necessary for the general AC setting. First, HPO does not minimize algorithm runtime, which is often the primary objective in general AC. instead, HPO aims at optimizing a solution quality metric, such as predictive accuracy. Of course, AC settings exist where runtime is not a configuration objective, such as when configuring metaheuristics to find the best possible solution in a given time budget. Second, HPO techniques lack a problem instance selection mechanism. Specifically, one configuration in HPO is run on all the instances of one dataset and the result is observed by the configurator. Note that this set should be seen as a single AC problem instance, i.e., the term “instance” is used differently between the HPO and general AC communities. In AC, the configuration needs to be
tested on a (sub)set of problem instances before the configurator can infer traits about its quality. Furthermore, HPO can be paired with algorithm selection, which is referred to as the combined algorithm selection and hyperparameter optimization problem (CASH) [Thornton et al., 2013].

**Automated Machine Learning (AutoML, CASH)** An algorithm selection component can be added to the HPO problem, resulting in the combined algorithm selection and hyperparameter optimization (CASH) problem as formalized by [Thornton et al., 2013]. Similar to HPO, the CASH problem can be classified as a sub-problem of algorithm configuration that is restricted to the domain of machine learning. Note that in the setting of AutoML, configurators typically face only a single AC problem instance in the form of a machine learning dataset. Due to this, we do not cover AutoML/CASH but instead refer the interested reader to comprehensive surveys [Elshawi et al., 2019; Zöller and Huber, 2021; Hutter et al., 2019].

### 3 Classification

To order and characterize AC settings and methods, we introduce a classification scheme that separately covers (1) the algorithm configuration setting and (2) the configurator itself. More precisely, the problem view describes the configuration setting a method is designed for. The problem view consists of eight subcategories with an emphasis on the properties of the problem and the interaction between the configurator and target algorithm. The configurator view consists of seven components that portray important aspects of a configurator. Both of these views are interconnected and complementary. Moreover, the configurator view can be interpreted as an answer to a problem setting, where specific features are added to the configurator as a response to the configuration setting. Existing classification schemes proposed in the literature until now [Huang et al., 2019; Eiben and Smit, 2011a; Eiben and Smit, 2011b; Stützle and López-Ibáñez, 2019; Eryoldas and Durmuşoğlu, 2021] focus solely on the configurator and ignore the problem setting. The proposed taxonomy allows for a description and characterization of methods by aggregating information in tuples. The scheme (especially the problem view) can also be used to derive new problem scenarios that have not been addressed before by combining different aspects in previously unseen ways.

#### 3.1 Problem View

The components of the problem view (Table 1) characterize a problem setting a configurator is meant for and therefore influence the configurator’s design. Figure 2 displays these interconnections and the communication between target algorithm and configurator, as well as the inputs a configurator receives. Note that, except for the objective function and external runtime setting, all other aspects are mutually exclusive, meaning that an unambiguous setting for a configurator exists. Furthermore, only the training setting and configuration scope are independent of the target algorithm. For further details on the different classes and their options, we refer to [Schede et al., 2022].

<table>
<thead>
<tr>
<th>Problem aspects</th>
<th>Options</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training setting</td>
<td>Offline</td>
<td>Realtime</td>
</tr>
<tr>
<td>Configuration scope</td>
<td>Set</td>
<td>Instance</td>
</tr>
<tr>
<td>Search space</td>
<td>Small discrete</td>
<td>Large discrete</td>
</tr>
<tr>
<td>Target algorithm objective type</td>
<td>Single-objective</td>
<td>Multi-objective</td>
</tr>
<tr>
<td>Objective function*</td>
<td>Solving time</td>
<td>Accuracy</td>
</tr>
<tr>
<td>Target algorithm observation time</td>
<td>During run</td>
<td>Post termination</td>
</tr>
<tr>
<td>Configuration adjustment</td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td>External runtime setting*</td>
<td>Limited</td>
<td>Infinite</td>
</tr>
</tbody>
</table>

* Options not mutually exclusive

#### 3.2 Configurator View

<table>
<thead>
<tr>
<th>Configurator aspect</th>
<th>Setting</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution quality guarantee</td>
<td>Heuristic</td>
<td>Proven</td>
</tr>
<tr>
<td>Surrogate models</td>
<td>Model-free</td>
<td>Model-based</td>
</tr>
<tr>
<td>Problem instance features</td>
<td>Feature-based</td>
<td>Feature-based</td>
</tr>
<tr>
<td>Target algorithm execution</td>
<td>Sequential</td>
<td>Parallel</td>
</tr>
<tr>
<td>Candidate output</td>
<td>Single configuration</td>
<td>Set configuration</td>
</tr>
<tr>
<td>Configurator objective</td>
<td>Single-objective</td>
<td>Multi-objective</td>
</tr>
<tr>
<td>Internal runtime setting</td>
<td>Limited</td>
<td>Infinite</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Policy</th>
</tr>
</thead>
</table>

Table 1: The problem view classification scheme.

Table 2: The configurator view classification scheme.
3.2 Configurator View
The configurator view (Table 2) characterizes algorithm configurators. The scheme does not cover concrete functionalities utilized by configurators such as intensification criteria or creation, selection and elimination of configurations. These functionalities are very difficult to characterize and classify, since for a single mechanism many options with only subtle differences may exist. We again refer the reader for full directions of the configurator view to [Schede et al., 2022].

4 Conclusion
Parameters are ubiquitous in modern optimization approaches and beyond, with all of the significant solvers for, e.g., MILP, SAT, or TSP problems containing parameters that influence their performance and need to be set by the user. AC frees the user from this tedious and error-prone task by automating the search for high-quality configurations. We presented an overview of the current state of AC methods. In particular, we provided two taxonomies for organizing AC approaches and put sub problems of AC into perceptive.

Acknowledgments
This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center “On-The-Fly Computing” (SFB 901/3 project no, 160364472) and by the research training group “Dataninja” (Trustworthy AI for Seamless Problem Solving: Next Generation Intelligence Joins Robust Data Analysis) funded by the German federal state of North Rhine-Westphalia.

References


