Sequential Model-Based Optimization for General Algorithm Configuration

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown

University of British Columbia, 2366 Main Mall, Vancouver BC, V6T 1Z4, Canada
{hutter,hoos,kevinlb}@cs.ubc.ca

Abstract. State-of-the-art algorithms for hard computational problems often expose many parameters that can be modified to improve empirical performance. However, manually exploring the resulting combinatorial space of parameter settings is tedious and tends to lead to unsatisfactory outcomes. Recently, automated approaches for solving this *algorithm configuration* problem have led to substantial improvements in the state of the art for solving various problems. One promising approach constructs explicit regression models to describe the dependence of target algorithm performance on parameter settings; however, this approach has so far been limited to the optimization of few numerical algorithm parameters on single instances. In this paper, we extend this paradigm for the first time to general algorithm configuration problems, allowing many categorical parameters and optimization for sets of instances. We experimentally validate our new algorithm configuration procedure by optimizing a local search and a tree search solver for the propositional satisfiability problem (SAT), as well as the commercial mixed integer programming (MIP) solver CPLEX. In these experiments, our procedure yielded state-of-the-art performance, and in many cases outperformed the previous best configuration approach.

1 Introduction

Algorithms for hard computational problems—whether based on local search or tree search—are often highly parameterized. Typical parameters in local search include neighbourhoods, tabu tenure, percentage of random walk steps, and perturbation and acceptance criteria in iterated local search. Typical parameters in tree search include decisions about preprocessing, branching rules, how much work to perform at each search node (*e.g.*, to compute cuts or lower bounds), which type of learning to perform, and when to perform restarts. As one prominent example, the commercial mixed integer programming solver IBM ILOG CPLEX has 76 parameters pertaining to its search strategy [1]. Optimizing the settings of such parameters can greatly improve performance, but doing so manually is tedious and often impractical.

Automated procedures for solving this *algorithm configuration* problem are useful in a variety of contexts. Their most prominent use case is to optimize parameters on a training set of instances from some application ("offline", as part of algorithm development) in order to improve performance when using the algorithm in practice ("online"). Algorithm configuration thus trades human time for machine time and automates a task that would otherwise be performed manually. End users of an algorithm can also apply

C.A. Coello Coello (Ed.): LION 5, LNCS 6683, pp. 507–523, 2011.

algorithm configuration procedures (*e.g.*, the automated tuning tool built into CPLEX versions 11 and above) to configure an existing algorithm for high performance on problem instances of interest.

The algorithm configuration problem can be formally stated as follows: given a parameterized algorithm A (the *target algorithm*), a set (or distribution) of problem instances I and a cost metric c, find parameter settings of A that minimize c on I. The cost metric c is often based on the runtime required to solve a problem instance, or, in the case of optimization problems, on the solution quality achieved within a given time budget. Various automated procedures have been proposed for solving this algorithm configuration problem. Existing approaches differ in whether or not explicit models are used to describe the dependence of target algorithm performance on parameter settings.

Model-free algorithm configuration methods are relatively simple, can be applied out-of-the-box, and have recently led to substantial performance improvements across a variety of constraint programming domains. This research goes back to the early 1990s [2, 3] and has lately been gaining momentum. Some methods focus on optimizing numerical (*i.e.*, either integer- or real-valued) parameters (see, *e.g.*, [4, 5]), while others also target categorical (*i.e.*, discrete-valued and unordered) domains [6, 7, 8, 9]. The most prominent configuration methods are the racing algorithm F-RACE [5] and our own iterated local search algorithm PARAMILS [7, 8]. A recent competitor is the genetic algorithm GGA [9]. F-RACE and its extensions have been used to optimize various high-performance algorithms, including iterated local search and ant colony optimization procedures for timetabling tasks and the travelling salesperson problem [6, 5]. Our own group has used PARAMILS to configure highly parameterized tree search [10] and local search solvers [11] for the propositional satisfiability problem (SAT), as well as several solvers for mixed integer programming (MIP), substantially advancing the state of the art for various types of instances. Notably, by optimizing the 76 parameters of CPLEX—the most prominent MIP solver—we achieved up to 50-fold speedups over the defaults and over the configuration returned by the CPLEX tuning tool [1].

While the progress in practical applications described above has been based on model-free optimization methods, recent progress in model-based approaches promises to lead to the next generation of algorithm configuration procedures. *Sequential model-based optimization (SMBO)* iterates between fitting models and using them to make choices about which configurations to investigate. It offers the appealing prospects of interpolating performance between observed parameter settings and of extrapolating to previously unseen regions of parameter space. It can also be used to quantify importance of each parameter and parameter interactions. However, being grounded in the "black-box function optimization" literature from statistics (see, e.g., [12]), SMBO has inherited a range of limitations inappropriate to the automated algorithm configuration setting. These limitations include a focus on deterministic target algorithms; use of costly initial experimental designs; reliance on computationally expensive models; and the assumption that all target algorithm runs have the same execution costs. Despite considerable recent advances [13, 14, 15], all published work on SMBO still has three key limitations that prevent its use for general algorithm configuration tasks: (1) it only supports numerical parameters; (2) it only optimizes target algorithm performance for

single instances; and (3) it lacks a mechanism for terminating poorly performing target algorithm runs early.

The main contribution of this paper is to remove the first two of these SMBO limitations, and thus to make SMBO applicable to general algorithm configuration problems with many categorical parameters and sets of benchmark instances. Specifically, we generalize four components of the SMBO framework and—based on them—define two novel SMBO instantiations capable of general algorithm configuration: the simple model-free Random Online Adaptive Racing (ROAR) procedure and the more sophisticated Sequential Model-based Algorithm Configuration (SMAC) method. These methods do not yet implement an early termination criterion for poorly performing target algorithm runs (such as, *e.g.*, PARAMILS's adaptive capping mechanism [8]); thus, so far we expect them to perform poorly on some configuration scenarios with large captimes. In a thorough experimental analysis for a wide range of 17 scenarios with small captimes (involving the optimization of local search and tree search SAT solvers, as well as the commercial MIP solver CPLEX), SMAC indeed compared favourably to the two most prominent approaches for general algorithm configuration: PARAMILS [7, 8] and GGA [9].

The remainder of this paper is structured as follows. Section 2 describes the SMBO framework and previous work on SMBO. Sections 3 and 4 generalize SMBO's components to tackle general algorithm configuration scenarios, defining ROAR and SMAC, respectively. Section 5 experimentally compares ROAR and SMAC to the existing state of the art in algorithm configuration. Section 6 concludes the paper.

2 Existing Work on Sequential Model-Based Optimization (SMBO)

Model-based optimization methods construct a regression model (often called a *response surface model*) that predicts performance and then use this model for optimization. *Sequential* model-based optimization (SMBO) iterates between fitting a model and gathering additional data based on this model. In the context of parameter optimization, the model is fitted to a training set $\{(\boldsymbol{\theta}_1, o_1), \ldots, (\boldsymbol{\theta}_n, o_n)\}$ where parameter configuration $\boldsymbol{\theta}_i = (\theta_{i,1}, \ldots, \theta_{i,d})$ is a complete instantiation of the target algorithm's d parameters and o_i is the target algorithm's observed performance when run with configuration $\boldsymbol{\theta}_i$. Given a new configuration $\boldsymbol{\theta}_{n+1}$, the model aims to predict its performance o_{n+1}.

SMBO has its roots in the statistics literature on experimental design for global continuous ("black-box") function optimization. Most notable is the efficient global optimization (EGO) algorithm by Jones et al. [12], which is, however, limited to optimizing continuous parameters for noise-free functions (*i.e.*, the performance of deterministic algorithms). Bartz-Beielstein et al. [13] were the first to use the EGO approach to optimize algorithm performance. Their sequential parameter optimization (SPO) toolbox–which has received considerable attention in the evolutionary algorithms community–provides many features that facilitate the manual analysis and optimization of algorithm parameters; it also includes an automated SMBO procedure for optimizing numerical parameters on single instances. We studied the components of this automated procedure, demonstrated that its intensification mechanism mattered most, and improved it in our SPO$^+$ algorithm [14]. In [15], we showed how to reduce the

Algorithm Framework 1: Sequential Model-Based Optimization (SMBO)
$\mathbf{R}$ keeps track of all target algorithm runs performed so far and their performances (*i.e.*, SMBO's training data $\{([\boldsymbol{\theta}_1, \boldsymbol{x}_1], o_1), \ldots, ([\boldsymbol{\theta}_n, \boldsymbol{x}_n], o_n)\}$), $\mathcal{M}$ is SMBO's model, $\vec{\boldsymbol{\Theta}}_{new}$ is a list of promising configurations, and t_{fit} and t_{select} are the runtimes required to fit the model and select configurations, respectively.

Input : Target algorithm A with parameter configuration space $\boldsymbol{\Theta}$; instance set Π; cost metric $\hat{c}$
Output : Optimized (incumbent) parameter configuration, $\boldsymbol{\theta}_{inc}$

$[\mathbf{R}, \boldsymbol{\theta}_{inc}] \leftarrow$ *Initialize*($\boldsymbol{\Theta}$, Π)
repeat
  $[\mathcal{M}, t_{fit}] \leftarrow$ *FitModel*($\mathbf{R}$)
  $[\vec{\boldsymbol{\Theta}}_{new}, t_{select}] \leftarrow$ *SelectConfigurations*($\mathcal{M}$, $\boldsymbol{\theta}_{inc}$, $\boldsymbol{\Theta}$)
  $[\mathbf{R}, \boldsymbol{\theta}_{inc}] \leftarrow$ *Intensify*($\vec{\boldsymbol{\Theta}}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathcal{M}$, $\mathbf{R}$, $t_{fit} + t_{select}$, Π, $\hat{c}$)
until *total time budget for configuration exhausted*
return $\boldsymbol{\theta}_{inc}$

overhead incurred by construction and use of response surface models via approximate GP models. We also eliminated the need for a costly initial design by interleaving randomly selected parameters throughout the optimization process, and by exploiting the fact that different algorithm runs take different amounts of time. The resulting time-bounded SPO variant, TB-SPO, is the first practical SMBO method for parameter optimization given a user-specified time budget. Although it was shown to significantly outperform PARAMILS in certain cases, it is still limited to the optimization of numerical algorithm parameters on single problem instances.

In Algorithm Framework 1, we give pseudocode for the time-bounded SMBO framework of which TB-SPO is an instantiation: in each iteration, it fits a model, selects a list of promising parameter configurations and performs target algorithm runs on (a subset of) these, until a given time bound is reached. This time bound is related to the combined overhead due to fitting the model and selecting promising configurations. In the following, we generalize the components of this algorithm framework, extending its scope to tackle general algorithm configuration problems with many categorical parameters and sets of benchmark instances.

3 Random Online Aggressive Racing (ROAR)

In this section, we first generalize SMBO's *Intensify* procedure to handle multiple instances, and then introduce ROAR, a very simple model-free algorithm configuration procedure based on this new intensification mechanism.

3.1 Generalization I: An Intensification Mechanism for Multiple Instances

A crucial component of any algorithm configuration procedure is the so-called *intensification* mechanism, which governs how many evaluations to perform with each configuration, and when to trust a configuration enough to make it the new current best known configuration (the *incumbent*). When configuring algorithms for sets of

Procedure 2: Intensify($\vec{\Theta}_{new}$, θ_{inc}, $\mathcal{M}$, $\mathbf{R}$, $t_{intensify}$, Π, $\hat{c}$)
$\hat{c}(\theta, \Pi')$ denotes the empirical cost of θ on the subset of instances $\Pi' \subseteq \Pi$, based on the runs in $\mathbf{R}$; *maxR* is a parameter, set to 2 000 in all our experiments

Input : Sequence of parameter settings to evaluate, $\vec{\Theta}_{new}$; incumbent parameter setting, θ_{inc}; model, $\mathcal{M}$; sequence of target algorithm runs, $\mathbf{R}$; time bound, $t_{intensify}$; instance set, Π; cost metric, $\hat{c}$
Output : Updated sequence of target algorithm runs, $\mathbf{R}$; incumbent parameter setting, θ_{inc}

for $i := 1, \ldots,$ *length*($\vec{\Theta}_{new}$) **do**
  $\theta_{new} \leftarrow \vec{\Theta}_{new}[i]$
  if $\mathbf{R}$ *contains less than* $maxR$ *runs with configuration* θ_{inc} **then**
    $\Pi' \leftarrow \{\pi' \in \Pi \mid \mathbf{R}$ contains less than or equal number of runs using θ_{inc} and π' than using θ_{inc} and any other $\pi'' \in \Pi\}$
    $\pi \leftarrow$ instance sampled uniformly at random from Π'
    $s \leftarrow$ seed, drawn uniformly at random
    $\mathbf{R} \leftarrow$ ExecuteRun($\mathbf{R}$, θ_{inc}, π, s)
  $N \leftarrow 1$
  while *true* **do**
    $S_{missing} \leftarrow \langle$instance, seed$\rangle$ pairs for which θ_{inc} was run before, but not θ_{new}
    $S_{torun} \leftarrow$random subset of $S_{missing}$ of size $\min(N, |S_{missing}|)$
    foreach $(\pi, s) \in S_{torun}$ **do** $\mathbf{R} \leftarrow$ ExecuteRun($\mathbf{R}$, θ_{new}, π, s)
    $S_{missing} \leftarrow S_{missing} \setminus S_{torun}$
    $\Pi_{common} \leftarrow$ instances for which we previously ran both θ_{inc} and θ_{new}
    if $\hat{c}(\theta_{new}, \Pi_{common}) > \hat{c}(\theta_{inc}, \Pi_{common})$ **then break**
    else if $S_{missing} = \emptyset$ **then** $\theta_{inc} \leftarrow \theta_{new}$; **break**
    else $N \leftarrow 2 \cdot N$
  if *time spent in this call to this procedure exceeds* $t_{intensify}$ *and* $i \geq 2$ **then break**
return $[\mathbf{R}, \theta_{inc}]$

instances, we also need to decide which instance to use in each run. To address this problem, we generalize TB-SPO's intensification mechanism. Our new procedure implements a variance reduction mechanism, reflecting the insight that when we compare the empirical cost statistics of two parameter configurations across multiple instances, the variance in this comparison is lower if we use the same N instances to compute both estimates.

Procedure 2 defines this new intensification mechanism more precisely. It takes as input a list of promising configurations, $\vec{\Theta}_{new}$, and compares them in turn to the current incumbent configuration until a time budget for this comparison stage is reached.[1] In each comparison of a new configuration, θ_{new}, to the incumbent, θ_{inc}, we first perform an additional run for the incumbent, using a randomly selected $\langle$instance, seed$\rangle$ combination. Then, we iteratively perform runs with θ_{new} (using a doubling scheme) until either θ_{new}'s empirical performance is worse than that of θ_{inc} (in which case we reject θ_{new}) or we performed as many runs for θ_{new} as for θ_{inc} and it is still at least as good as θ_{inc} (in which case we change the incumbent to θ_{new}). The $\langle$instance, seed$\rangle$ combi-

[1] If that budget is already reached after the first configuration in $\vec{\Theta}_{new}$, one more configuration is used; see the last paragraph of Section 4.3 for an explanation why.

nations for $\boldsymbol{\theta}_{new}$ are sampled uniformly at random from those on which the incumbent has already run. Similar to the FOCUSEDILS algorithm [7, 8], $\boldsymbol{\theta}_{inc}$ and $\boldsymbol{\theta}_{new}$ are always compared using only instances on which they have both been run. However, every comparison in Procedure 2 is based on a *different* randomly selected subset of instances and seeds, while FOCUSEDILS's Procedure "better" uses a fixed ordering to which it can be very sensitive.

3.2 Defining ROAR

We now define Random Online Aggressive Racing (ROAR), a simple model-free instantiation of the general SMBO framework (see Algorithm Framework 1).[2] This surprisingly effective method selects parameter configurations uniformly at random and iteratively compares them against the current incumbent using our new intensification mechanism. We consider ROAR to be a racing algorithm, because it runs each candidate configuration only as long as necessary to establish whether it is competitive. It gets its name because the set of candidates is selected at *random*, each candidate is accepted or rejected *online*, and we make this online decision *aggressively*, before enough data has been gathered to support a statistically significant conclusion. More formally, as an instantiation of the SMBO framework, ROAR is completely specified by the four components *Initialize*, *FitModel*, *SelectConfigurations*, and *Intensify*. *Initialize* performs a single run with the target algorithm's default parameter configuration (or a random configuration if no default is available) on an instance selected uniformly at random. Since ROAR is model-free, its *FitModel* procedure simply returns a constant model which is never used. *SelectConfigurations* returns a single configuration sampled uniformly at random from the parameter space, and *Intensify* is as described in Procedure 2.

4 Sequential Model-Based Algorithm Configuration (SMAC)

In this section, we introduce our second, more sophisticated instantiation of the general SMBO framework: Sequential Model-based Algorithm Configuration (SMAC). SMAC can be understood as an extension of ROAR that selects configurations based on a model rather than uniformly at random. It instantiates *Initialize* and *Intensify* in the same way as ROAR. Here, we discuss the new model class we use in SMAC to support categorical parameters and multiple instances (Sections 4.1 and 4.2, respectively); then, we describe how SMAC uses its models to select promising parameter configurations (Section 4.3).

4.1 Generalization II: Models for Categorical Parameters

The models in all existing SMBO methods of which we are aware are limited to numerical parameters. In this section, we discuss the new model class SMAC uses to also handle *categorical* parameters.

[2] We previously considered random sampling approaches based on less powerful intensification mechanisms; see, *e.g.*, RANDOM* defined in [15].

SMAC's models are based on random forests [16], a standard machine learning tool for regression and classification.[3] Random forests are collections of regression trees, which are similar to decision trees but have real values (here: target algorithm performance values) rather than class labels at their leaves. Regression trees are known to perform well for categorical input data; indeed, they have already been used for modeling the performance of heuristic algorithms (e.g., [18, 19]). Random forests share this benefit and typically yield more accurate predictions [16]; they also allow us to quantify our uncertainty in a given prediction. We construct a random forest as a set of B regression trees, each of which is built on n data points randomly sampled with repetitions from the entire training data set $\{(\boldsymbol{\theta}_1, o_1), \ldots, (\boldsymbol{\theta}_n, o_n)\}$. At each split point of each tree, a random subset of $\lceil d \cdot p \rceil$ of the d algorithm parameters is considered eligible to be split upon; the split ratio p is a parameter, which we left at its default of $p = 5/6$. A further parameter is n_{min}, the minimal number of data points required to be in a node if it is to be split further; we use the standard value $n_{min} = 10$. Finally, we set the number of trees to $B = 10$ to keep the computational overhead small.[4] We compute the random forest's predictive mean $\mu_{\boldsymbol{\theta}}$ and variance $\sigma^2_{\boldsymbol{\theta}}$ for a new configuration $\boldsymbol{\theta}$ as the empirical mean and variance of its individual trees' predictions for $\boldsymbol{\theta}$.

Model fit can often be improved by transforming the cost metric. In this paper, we focus on minimizing algorithm runtime. Previous work on predicting algorithm runtime has found that logarithmic transformations substantially improve model quality [20] and we thus use log-transformed runtime data throughout this paper; that is, for runtime r_i, we use $o_i = \ln(r_i)$. (SMAC can also be applied to optimize other cost metrics, such as the solution quality an algorithm obtains in a fixed runtime; other transformations may prove more efficient for other metrics.) However, we note that in some models such transformations implicitly change the cost metric users aim to optimize [17]. We avoid this problem in our random forests by computing the prediction in the leaf of a tree by "untransforming" the data, computing the user-defined cost metric, and then transforming the result again.

4.2 Generalization III: Models for Sets of Problem Instances

There are several possible ways to extend SMBO's models to handle multiple instances. Most simply, one could use a fixed set of N instances for every evaluation of the target algorithm run, reporting aggregate performance. However, there is no good fixed choice for N: small N leads to poor generalization to test data, while large N leads to a

[3] In principle, other model families can also be used. Notably, one might consider Gaussian processes (GPs), or the projected process (PP) approximation to GPs we used in TB-SPO [15]. Although GPs are canonically defined only for numerical parameters, they can be extended to categorical parameters by changing the kernel function. We defined such a kernel function, based on the weighted Hamming distance between two parameter configurations. However, there is a more significant obstacle to using GPs to support general algorithm configuration: response variable transformations distort the GP cost metric, which is particularly problematic for multi-instance models. Further information, including the definition of the weighted Hamming distance kernel function, can be found in the extended version of this paper [17].

[4] An optimization of these three parameters might improve performance further. We plan on studying this in the context of an application of SMAC to optimizing its own parameters.

prohibitive N-fold slowdown in the cost of each evaluation. (This is the same problem faced by the PARAMILS instantiation BASICILS(N) [7].) Instead, we explicitly integrate information about the instances into our response surface models. Given a vector of *features* $\boldsymbol{x}_i$ describing each training problem instance $\pi_i \in \Pi$, we learn a joint model that predicts algorithm runtime for combinations of parameter configurations and instance features. We then aggregate these predictions across instances.

Instance Features. Existing work on empirical hardness models [21] has demonstrated that it is possible to predict algorithm runtime based on features of a given problem instance. Most notably, such predictions have been exploited to construct portfolio-based algorithm selection mechanisms, such as SATzilla [20]. For SAT instances in the form of CNF formulae, we used 126 features including features based on graph representations of the instance, an LP relaxation, DPLL probing, local search probing, clause learning, and survey propagation. For MIP instances we computed 39 features, including features based on graph representations, an LP relaxation, the objective function, and the linear constraint matrix. Both sets of features are detailed in the extended version of this paper [17]. To reduce the computational complexity of learning, we applied *principal component analysis* (see, *e.g.*, [22]), to project the feature matrix into a lower-dimensional subspace spanned by the seven orthogonal vectors along which it has maximal variance. For new domains, for which no features have yet been defined, SMAC can still be applied with an empty feature set or simple domain-independent features, such as instance size or the performance of the algorithm's default setting (which, based on preliminary experiments, seems to be a surprisingly effective feature). Note that in contrast to per-instance approaches, instance features are only needed for the *training* instances: the end result of algorithm configuration is a single parameter configuration that is used without a need to compute features for test instances.

Predicting Performance Across Instances. So far, we have discussed models trained on pairs $(\boldsymbol{\theta}_i, o_i)$ of parameter configurations $\boldsymbol{\theta}_i$ and their observed performance o_i. Now, we extend this data to include instance features. Let $\boldsymbol{x}_i$ denote the vector of features for the instance used in the ith target algorithm run. Concatenating parameter values, $\boldsymbol{\theta}_i$, and instance features, $\boldsymbol{x}_i$, into one input vector yields the training data $\{([\boldsymbol{\theta}_1, \boldsymbol{x}_1], o_1), \ldots, ([\boldsymbol{\theta}_n, \boldsymbol{x}_n], o_n)\}$. From this data, we learn a model that takes as input a parameter configuration $\boldsymbol{\theta}$ and predicts performance across all training instances. To achieve this, we do not need to change random forest construction: all input dimensions are handled equally, regardless of whether they refer to parameter values or instance features. The prediction procedure changes as follows: within each tree, we first predict performance for the combinations of the given configuration and each instance; next, we combine these predictions with the user-defined cost metric (*e.g.*, arithmetic mean runtime across instances); finally, we compute means and variances across trees.

4.3 Generalization IV: Using the Model to Select Promising Configurations in Large Mixed Numerical/Categorical Configuration Spaces

The *SelectConfiguration* component in SMAC uses the model to select a list of promising parameter configurations. To quantify how promising a configuration $\boldsymbol{\theta}$ is, it uses the model's predictive distribution for $\boldsymbol{\theta}$ to compute its *expected* positive *improvement*

($EI(\boldsymbol{\theta})$) [12] over the best configuration seen so far (the *incumbent*). $EI(\boldsymbol{\theta})$ is large for configurations $\boldsymbol{\theta}$ with low predicted cost and for those with high predicted uncertainty; thereby, it offers an automatic tradeoff between exploitation (focusing on known good parts of the space) and exploration (gathering more information in unknown parts of the space). Specifically, we use the $E[I_{\text{exp}}]$ criterion introduced in [14] for log-transformed costs; given the predictive mean $\mu_{\boldsymbol{\theta}}$ and variance $\sigma^2_{\boldsymbol{\theta}}$ of the log-transformed cost of a configuration $\boldsymbol{\theta}$, this is defined as

$$EI(\boldsymbol{\theta}) := E[I_{\text{exp}}(\boldsymbol{\theta})] = f_{min}\Phi(v) - e^{\frac{1}{2}\sigma^2_{\boldsymbol{\theta}}+\mu_\theta} \cdot \Phi(v - \sigma_{\boldsymbol{\theta}}), \tag{1}$$

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 $\boldsymbol{\theta}_{inc}$.[5]

Having defined $EI(\theta)$, we must still decide how to identify configurations $\boldsymbol{\theta}$ with large $EI(\boldsymbol{\theta})$. This amounts to a maximization problem across parameter configuration space. Previous SMBO methods [13, 14, 15] simply applied random sampling for this task (in particular, they evaluated EI for $10\,000$ random samples), which is unlikely to be sufficient in high-dimensional configuration spaces, especially if promising configurations are sparse. To gather a set of promising configurations with low computational overhead, we perform a simple multi-start local search and consider all resulting configurations with locally maximal EI.[6] This search is similar in spirit to PARAMILS [7, 8], but instead of algorithm performance it optimizes $EI(\boldsymbol{\theta})$ (see Equation 1), which can be evaluated based on the model predictions $\mu_{\boldsymbol{\theta}}$ and $\sigma^2_{\boldsymbol{\theta}}$ without running the target algorithm. More concretely, the details of our local search are as follows. We compute EI for all configuations used in previous target algorithm runs, pick the ten configurations with maximal EI, and initialize a local search at each of them. To seamlessly handle mixed categorical/numerical parameter spaces, we use a randomized one-exchange neighbourhood, including the set of all configurations that differ in the value of exactly one discrete parameter, as well as four random neighbours for each numerical parameter. In particular, we normalize the range of each numerical parameter to [0,1] and then sample four "neighbouring" values for numerical parameters with current value v from a univariate Gaussian distribution with mean v and standard deviation 0.2, rejecting new values outside the interval [0,1]. Since batch model predictions (and thus batch EI computations) for a set of N configurations are much cheaper than separate predictions for N configurations, we use a best improvement search, evaluating EI for all neighbours at once; we stop each local search once none of the neighbours has larger EI. Since SMBO sometimes evaluates many configurations per iteration and because batch EI computations are cheap, we simply compute EI for an additional $10\,000$ randomly-sampled configurations; we then sort all $10\,010$ configurations in descending order of EI. (The ten results of local search typically had larger EI than all randomly sampled configurations.)

[5] In TB-SPO [15], we used $f_{min} = \mu(\boldsymbol{\theta}_{inc}) + \sigma(\boldsymbol{\theta}_{inc})$. However, we now believe that setting f_{min} to the empirical mean performance of $\boldsymbol{\theta}_{inc}$ yields better performance overall.

[6] We plan to investigate better mechanisms in the future. However, we note that the best problem formulation is not obvious, since we desire a *diverse* set of configurations with high EI.

Having selected this list of 10 010 configurations based on the model, we interleave randomly-sampled configurations in order to provide unbiased training data for future models. More precisely, we alternate between configurations from the list and additional configurations sampled uniformly at random. Since *Intensify* always compares at least two configurations against the current incumbent, at least one randomly sampled configuration is evaluated in every iteration of SMBO. In finite configuration spaces, thus, each configuration has a positive probability of being selected in each iteration. In combination with the fact that *Intensify* increases the number of runs used to evaluate each configuration unboundedly, this allows us to prove that SMAC (and ROAR) eventually converge to the optimal configuration when using consistent estimators of the user-defined cost metric.[7] The proof is very simple and uses the same arguments as a previous proof about FocusedILS (see [8]); we omit it here and refer the reader to the extended version of this paper [17].

5 Experimental Evaluation

We now compare the performance of SMAC, ROAR, TB-SPO [15], GGA [9], and PARAMILS (in particular, FOCUSEDILS 2.3) [8] for a range of configuration scenarios that involve minimizing the runtime of SAT and MIP solvers. In principle, our ROAR and SMAC methods also apply to optimizing other cost metrics, such as the solution quality an algorithm can achieve in a fixed time budget; we plan on studying their empirical performance for this case in the near future.

5.1 Experimental Setup

Configuration scenarios. We considered a diverse set of 17 algorithm configuration problem instances (so-called *configuration scenarios*) that had been used previously to analyze PARAMILS [8, 1] and TB-SPO [15].[8] These scenarios involve the configuration of the local search SAT solver SAPS (4 parameters), the tree search solver SPEAR (26 parameters), and the most widely used commercial mixed integer programming (MIP) solver, IBM ILOG CPLEX (76 parameters); references for these algorithms, as well as details on their parameter spaces, are given in the extended version of this paper [17]. In all 17 configuration scenarios, we terminated target algorithm runs at $\kappa_{max} = 5$ seconds, the same per-run captime used in previous work for these scenarios. In previous work, we have also applied PARAMILS to optimize MIP solvers with very large per-run captimes (up to $\kappa_{max} = 10\,000$s), and obtained better results than the CPLEX tuning tool [1]. We believe that for such large captimes, an adaptive capping mechanism, such as the one implemented in ParamILS [8], is essential; we are currently working on

[7] This proof does not cover continuous parameters, since they lead to infinite configuration spaces; in that case, we would require additional smoothness assumptions to prove convergence.

[8] All instances we used are available at http://www.cs.ubc.ca/labs/beta/Projects/AAC

integrating such a mechanism into SMAC.[9] In this paper, to study the remaining components of SMAC, we only use scenarios with small captimes of 5s. In order to enable a fair comparison with GGA, we changed the optimization objective of all 17 scenarios from the original PAR-10 (*penalized average runtime*, counting timeouts at κ_{max} as $10 \cdot \kappa_{max}$, which is not supported by GGA) to simple average runtime (PAR-1, counting timeouts at κ_{max} as κ_{max}). [10] However, one difference remains: we minimize the runtime reported by the target algorithm, but GGA can only minimize its own measurement of target algorithm runtime, including (sometimes large) overheads for reading in the instance.

Parameter transformations. Some numerical parameters naturally vary on a non-uniform scale (*e.g.*, a parameter θ with an interval $[100, 1600]$ that we discretized to the values $\{100, 200, 400, 800, 1600\}$ for use in PARAMILS). We transformed such parameters to a domain in which they vary more uniformly (*e.g.*, $\log(\theta) \in [\log(100), \log(1600)]$), un-transforming the parameter values for each call to the target algorithm.

Comparing configuration procedures. We performed 25 runs of each configuration procedure on each configuration scenario. For each such run r_i, we computed *test performance* t_i as follows. First, we extracted the incumbent configuration $\boldsymbol{\theta}_{inc}$ at the point the configuration procedure exhausted its time budget; SMAC's overhead due to the construction and use of models were counted as part of this budget. Next, in an offline evaluation step using the same per-run cutoff time as during training, we measured the mean runtime t_i across 1 000 independent test runs of the target algorithm parameterized by $\boldsymbol{\theta}_{inc}$. In the case of multiple-instance scenarios, we used a test set of previously unseen instances. For a given scenario, this resulted in test performances $t_1, \ldots, t_{25}$ for each configuration procedure. We report medians across these 25 values, visualize their variance in boxplots, and perform a Mann-Whitney U test to check for significant differences between configuration procedures. We ran GGA through HAL [23], using parameter settings recommended by GGA's author, Kevin Tierney, in e-mail communication: we set the population size to 70, the number of generations to 100, the number of runs to perform in the first generation to 5, and the number of runs to perform in the last generation to 70. We used default settings for FOCUSEDILS 2.3, including aggressive capping. We note that in a previous comparison [9] of GGA and FOCUSEDILS, capping was disabled in FOCUSEDILS; this explains its poor performance there and its better performance here.

[9] In fact, preliminary experiments for configuration scenario CORLAT (from [1], with $\kappa_{max} = 10\,000$s) highlight the importance of developing an adaptive capping mechanism for SMAC: *e.g.*, in one of SMAC's run, it only performed 49 target algorithm runs, with 15 of them timing out after $\kappa_{max} = 10\,000$s, and another 3 taking over 5 000 seconds each. Together, these runs exceeded the time budget of 2 CPU days (172 800 seconds), despite the fact that all of them could have safely been cut off after less than 100 seconds. As a result, for scenario CORLAT, SMAC performed a factor of 3 worse than PARAMILS with $\kappa_{max} = 10\,000$s. On the other hand, SMAC can sometimes achieve strong performance even with relatively high captimes; *e.g.*, on CORLAT with $\kappa_{max} = 300$s, SMAC outperformed PARAMILS by a factor of 1.28.

[10] Using PAR-10 to compare the remaining configurators, our qualitative results did not change.

With the exception of FOCUSEDILS, all of the configuration procedures we study here support numerical parameters without a need for discretization. We present results both for the mixed numerical/categorical parameter space these methods search, and—to enable a direct comparison to FOCUSEDILS—for a fully discretized configuration space.

Computational environment. We conducted all experiments on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running OpenSuSE Linux 11.1. We measured runtimes as CPU time on these reference machines.

5.2 Experimental Results for Single Instance Scenarios

In order to evaluate our new general algorithm configuration procedures ROAR and SMAC one component at a time, we first evaluated their performance for optimizing the continuous parameters of SAPS and the mixed numerical/categorical parameters of SPEAR on single SAT instances; multi-instance scenarios are studied in the next section. To enable a comparison with our previous SMBO instantiation TB-SPO, we used the 6 configuration scenarios introduced in [15], which aim to minimize SAPS's runtime on 6 single SAT-encoded instances, 3 each from quasigroup completion (QCP) and small world graph colouring (SWGCP). We also used 5 similar new configuration scenarios, which aim to minimize SPEAR's runtime for 5 single SAT-encoded instances, 2 from a hard distribution of IBM bounded model checking (IBM) and 3 from software verification (SWV). For more information and references for these instances, please see [17]. The time budget for each algorithm configuration run was 30 CPU minutes, exactly following [15].

The model-based approaches SMAC and TB-SPO performed best in this comparison, followed by ROAR, FOCUSEDILS, and GGA. Table 1 shows the results achieved by each of the configuration procedures, for both the full parameter configuration space (which includes numerical parameters) and the discretized version we made for use with FOCUSEDILS. For the special case of single instances and a small number of all-numerical parameters, SMAC and TB-SPO are very similar, and both performed best.[11] While TB-SPO does not apply in the remaining configuration scenarios, our more general SMAC method achieved the best performance in all of them. ROAR performed well for small but not for large configuration spaces: it was among the best (*i.e.*, best or not significantly different from the best) in most of the SAPS scenarios (4 parameters) but only for one of the SPEAR scenarios (26 parameters). Both GGA and FOCUSEDILS performed slightly worse than ROAR for the SAPS scenarios, and slightly (but statistically significantly) worse than SMAC for most SPEAR configuration scenarios. Figure 1 visualizes each configurator's 25 test performances for all scenarios. We note that SMAC

[11] In fact, in 1 of the 6 scenarios for which TB-SPO is applicable, it performed better than SMAC. This is because for all-numerical parameters, projected process (PP) models performed better than random forest (RF) models. In further experiments (not reported here, see [17]), we evaluated a version of SMAC based on PP instead of RF models; its median performance was slightly better than TB-SPO's, but the two were statistically indistinguishable in all 6 scenarios. With categorical parameters, SMAC performed better with the RF models we use here.

Table 1. Comparison of algorithm configuration procedures for optimizing parameters on single problem instances. We performed 25 independent runs of each configuration procedure and report the median of the 25 test performances (mean runtimes across 1 000 target algorithm runs with the found configurations). We bold-faced entries for configurators that are not significantly worse than the best configurator for the respective configuration space, based on a Mann-Whitney U test. The symbol "—" denotes that the configurator does not apply for this configuration space.

Scenario	Unit	Full configuration space					Discretized configuration space				
		SMAC	TB-SPO	ROAR	F-ILS	GGA	SMAC	TB-SPO	ROAR	F-ILS	GGA
SAPS-QCP-MED	$[\cdot 10^{-2}s]$	4.70	**4.58**	4.72	—	6.28	**5.27**	—	**5.25**	5.50	6.24
SAPS-QCP-Q075	$[\cdot 10^{-1}s]$	**2.29**	**2.22**	2.34	—	2.74	**2.87**	—	**2.92**	**2.91**	**2.98**
SAPS-QCP-Q095	$[\cdot 10^{-1}s]$	**1.37**	**1.35**	1.55	—	1.75	**1.51**	—	**1.57**	**1.57**	1.95
SAPS-SWGCP-MED	$[\cdot 10^{-1}s]$	**1.61**	**1.63**	**1.70**	—	2.48	**2.54**	—	**2.58**	2.57	2.71
SAPS-SWGCP-Q075	$[\cdot 10^{-1}s]$	**2.11**	**2.48**	2.32	—	3.19	**3.26**	—	3.38	3.55	3.55
SAPS-SWGCP-Q095	$[\cdot 10^{-1}s]$	**2.36**	**2.69**	**2.49**	—	**3.13**	**3.65**	—	**3.79**	**3.75**	**3.77**
SPEAR-IBM-Q025	$[\cdot 10^{-1}s]$	**6.24**	—	6.31	—	6.33	**6.21**	—	6.30	6.31	6.30
SPEAR-IBM-MED	$[\cdot 10^{0}\ s]$	**3.28**	—	**3.36**	—	**3.35**	**3.16**	—	3.38	3.47	3.84
SPEAR-SWV-MED	$[\cdot 10^{-1}s]$	**6.04**	—	6.11	—	6.14	**6.05**	—	6.14	6.11	6.15
SPEAR-SWV-Q075	$[\cdot 10^{-1}s]$	**5.76**	—	5.88	—	5.83	**5.76**	—	5.89	5.88	5.84
SPEAR-SWV-Q095	$[\cdot 10^{-1}s]$	**8.38**	—	8.55	—	8.47	**8.42**	—	8.53	8.58	8.49

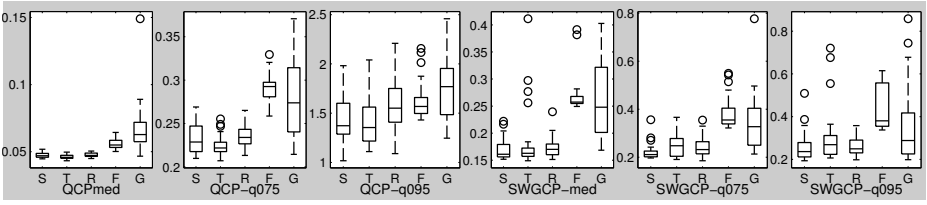
and ROAR often yielded more robust results than FOCUSEDILS and GGA: for many scenarios some of the 25 FOCUSEDILS and GGA runs did very poorly.

Our new SMAC and ROAR methods were able to explore the full configuration space, which sometimes led to substantially improved performance compared to the discretized configuration space PARAMILS is limited to. Comparing the left *vs* the right side of Table 1, we note that the SAPS discretization (the same we used to optimize SAPS with PARAMILS in previous work [7, 8]) left substantial room for improvement when exploring the full space: roughly 1.15-fold and 1.55-fold speedups on the QCP and SWGCP instances, respectively. GGA did not benefit as much from being allowed to explore the full configuration space for the SAPS scenarios; however, in one of the SPEAR scenarios (SPEAR-IBM-MED), it did perform 1.15 times better for the full space (albeit still worse than SMAC).

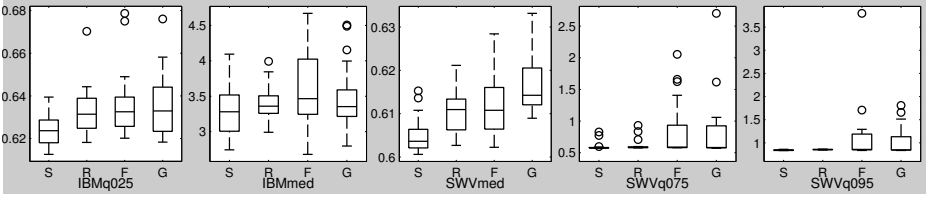
5.3 Experimental Results for General Multi-instance Configuration Scenarios

We now compare the performance of SMAC, ROAR, GGA, and FOCUSEDILS on six general algorithm configuration tasks that aim to minimize the mean runtime of SAPS, SPEAR, and CPLEX for various sets of instances. These are the 5 BROAD configuration scenarios used in [8] to evaluate PARAMILS's performance, plus one further CPLEX scenario, and we used the same time budget of 5 hours per configuration run.

Overall, SMAC performed best in this comparison: as shown in Table 2 its performance was among the best (*i.e.*, statistically indistinguishable from the best) in all 6 configuration scenarios, for both the discretized and the full configuration spaces. Our simple ROAR method performed surprisingly well, indicating the importance of the intensification mechanism: it was among the best in 2 of the 6 configuration scenarios for either version of the configuration space. However, it performed substantially worse than the best approaches for configuring CPLEX—the algorithm with the largest configuration space; we note that ROAR's random sampling approach lacks the guidance offered by either FOCUSEDILS's local search or SMAC's response surface model.



(a) SAPS (4 continuous parameters)



(b) SPEAR (26 parameters; 12 of them continuous and 4 integral)

Fig. 1. Visual comparison of configuration procedures' performance for setting SAPS and SPEAR's parameters for single instances. For each configurator and scenario, we show boxplots for the 25 test performances underlying Table 1, for the full configuration space (discretized for FOCUSEDILS). 'S' stands for SMAC, 'T' for TB-SPO, 'R' for ROAR, 'F' for FOCUSEDILS, and 'G' for GGA.

GGA performed slightly better for optimizing CPLEX than ROAR, but also significantly worse than either FOCUSEDILS or SMAC. Figure 2 visualizes the performance each configurator achieved for all 6 scenarios. We note that—similarly to the single instance cases—the results of SMAC were often more robust than those of FOCUSEDILS and GGA.

Although the performance improvements achieved by our new methods might not appear large in absolute terms, it is important to remember that algorithm configuration is an optimization problem, and that the ability to tease out the last few percent of improvement often distinguishes good algorithms. We expect the difference between configuration procedures to be clearer in scenarios with larger per-instance runtimes. In order to handle such scenarios effectively, we believe that SMAC will require an adaptive capping mechanism similar to the one we introduced for PARAMILS [8]; we are actively working on integrating such a mechanism with SMAC's models.

As in the single-instance case, for some configuration scenarios, SMAC and ROAR achieved much better results when allowed to explore the full space rather than FOCUSEDILS's discretized search space. Speedups for SAPS were similar to those observed in the single-instance case (about 1.15-fold for SAPS-QCP and 1.65-fold for SAPS-SWGCP), but now we also observed a 1.17-fold improvement for SPEAR-QCP. In contrast, GGA actually performed worse for 4 of the 6 scenarios when allowed to explore the full space.

Table 2. Comparison of algorithm configuration procedures for benchmarks with multiple instances. We performed 25 independent runs of each configuration procedure and report the median of the 25 test performances (mean runtimes across 1 000 target algorithm runs with the found configurations on a test set disjoint from the training set). We bold-face entries for configurators that are not significantly worse than the best configurator for the respective configuration space. We also list performance of the default configuration, and of the configuration found by the CPLEX tuning tool (see [1]); note that on the test set this can be worse than the default.

Scenario	Unit	Default	CPLEX Tuning Tool	Full configuration space				Discretized configuration space			
				SMAC	ROAR	F-ILS	GGA	SMAC	ROAR	F-ILS	GGA
SAPS-QCP	$[\cdot 10^{-1}s]$	11.8	—	**7.05**	7.52	—	7.84	**7.65**	**7.65**	**7.62**	**7.59**
SAPS-SWGCP	$[\cdot 10^{-1}s]$	25.0	—	**1.77**	**1.8**	—	2.82	**2.94**	3.01	**2.91**	3.04
SPEAR-QCP	$[\cdot 10^{-1}s]$	3.27	—	**1.65**	1.84	—	2.21	**1.93**	2.01	2.08	2.01
SPEAR-SWGCP	$[\cdot 10^{0}\ s]$	1.62	—	**1.16**	**1.16**	—	1.17	**1.16**	**1.16**	1.18	1.18
CPLEX-REGIONS100	$[\cdot 10^{-1}s]$	7.40	8.60	**3.45**	6.67	—	4.37	**3.50**	7.23	**3.23**	3.98
CPLEX-MIK	$[\cdot 10^{0}\ s]$	4.87	3.56	**1.20**	2.81	—	3.42	**1.24**	3.11	2.71	3.32

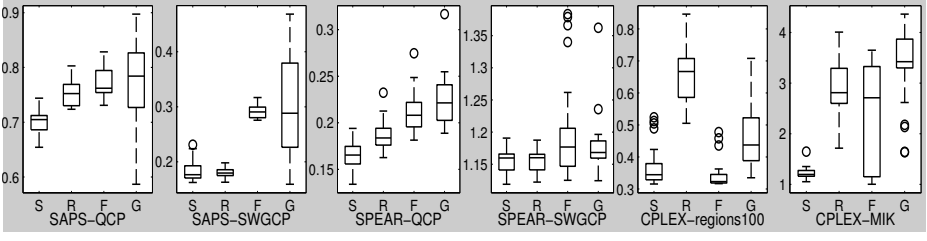


Fig. 2. Visual comparison of configuration procedures for general algorithm configuration scenarios. For each configurator and scenario, we show boxplots for the runtime data underlying Table 2, for the full configuration space (discretized for FOCUSEDILS). 'S' stands for SMAC, 'R' for ROAR, 'F' for FOCUSEDILS, and 'G' for GGA.

6 Conclusion

In this paper, we extended a previous line of work on sequential model-based optimization (SMBO) to tackle general algorithm configuration problems. SMBO had previously been applied only to the optimization of algorithms with numerical parameters on single problem instances. Our work overcomes both of these limitations, allowing categorical parameters and configuration for sets of problem instances. The four technical advances that made this possible are (1) a new intensification mechanism that employs blocked comparisons between configurations; an alternative class of response surface models, random forests, to handle (2) categorical parameters and (3) multiple instances; and (4) a new optimization procedure to select the most promising parameter configuration in a large mixed categorical/numerical space.

We presented empirical results for the configuration of two SAT algorithms (one local search, one tree search) and the commercial MIP solver CPLEX on a total of 17 configuration scenarios with small per-run captimes for each target algorithm run. Overall, our new SMBO procedure SMAC yielded statistically significant—albeit sometimes small—improvements over all of the other approaches on several configuration

scenarios, and never performed worse. In contrast to FOCUSEDILS, our new methods are also able to search the full (non-discretized) configuration space, which led to further substantial improvements for several configuration scenarios. We note that our new intensification mechanism enabled even ROAR, a simple model-free approach, to perform better than previous general-purpose configuration procedures in many cases; ROAR only performed poorly for optimizing CPLEX, where good configurations are sparse. SMAC yielded further improvements over ROAR and—most importantly—also state-of-the-art performance for the configuration of CPLEX.

In future work, we plan to improve SMAC to better handle configuration scenarios with large per-run captimes for each target algorithm run; specifically, we plan to integrate PARAMILS's adaptive capping mechanism into SMAC, which will require an extension of SMACs models to handle the resulting partly *censored* data. While in this paper we aimed to find a single configuration with overall good performance, we also plan to use SMAC's models to determine good configurations on a per-instance basis. Finally, we plan to use these models to characterize the importance of individual parameters and their interactions, and to study interactions between parameters and instance features.

Acknowledgements

We thank Kevin Murphy for many useful discussions on the modelling aspect of this work. Thanks also to Chris Fawcett and Chris Nell for help with running GGA through HAL, to Kevin Tierney for help with GGA's parameters, and to James Styles and Mauro Vallati for comments on an earlier draft of this paper. We gratefully acknowledge support from a postdoctoral research fellowship by the Canadian Bureau for International Education (FH), support from NSERC through HH's and KLB's respective discovery grants, and from the MITACS NCE through a seed project grant.

References

[1] Hutter, F., Hoos, H.H., Leyton-Brown, K.: Automated configuration of mixed integer programming solvers. In: Lodi, A., Milano, M., Toth, P. (eds.) CPAIOR 2010. LNCS, vol. 6140, pp. 186–202. Springer, Heidelberg (2010)

[2] Minton, S., Johnston, M.D., Philips, A.B., Laird, P.: Minimizing conflicts: A heuristic repair method for constraint-satisfaction and scheduling problems. AIJ 58(1), 161–205 (1992)

[3] Gratch, J., Dejong, G.: Composer: A probabilistic solution to the utility problem in speed-up learning. In: Proc. of AAAI 1992, pp. 235–240 (1992)

[4] Adenso-Diaz, B., Laguna, M.: Fine-tuning of algorithms using fractional experimental design and local search. Operations Research 54(1), 99–114 (2006)

[5] Birattari, M., Yuan, Z., Balaprakash, P., Stützle, T.: F-race and iterated F-race: an overview. In: Empirical Methods for the Analysis of Optimization Algorithms. Springer, Berlin (2010)

[6] Birattari, M., Stützle, T., Paquete, L., Varrentrapp, K.: A racing algorithm for configuring metaheuristics. In: Proc. of GECCO 2002, pp. 11–18 (2002)

[7] Hutter, F., Hoos, H.H., Stützle, T.: Automatic algorithm configuration based on local search. In: Proc. of AAAI 2007, pp. 1152–1157 (2007)

[8] Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. JAIR 36, 267–306 (2009)

[9] Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 142–157. Springer, Heidelberg (2009)
[10] Hutter, F., Babić, D., Hoos, H.H., Hu, A.J.: Boosting Verification by Automatic Tuning of Decision Procedures. In: Proc. of FMCAD 2007, pp. 27–34 (2007)
[11] KhudaBukhsh, A., Xu, L., Hoos, H.H., Leyton-Brown, K.: SATenstein: Automatically building local search SAT solvers from components. In: Proc. of IJCAI 2009 (2009)
[12] Jones, D.R., Schonlau, M., Welch, W.J.: Efficient global optimization of expensive black box functions. Journal of Global Optimization 13, 455–492 (1998)
[13] Bartz-Beielstein, T., Lasarczyk, C., Preuss, M.: Sequential parameter optimization. In: Proc. of CEC 2005, pp. 773–780. IEEE Press, Los Alamitos (2005)
[14] Hutter, F., Hoos, H.H., Leyton-Brown, K., Murphy, K.P.: An experimental investigation of model-based parameter optimisation: SPO and beyond. In: Proc. of GECCO 2009 (2009)
[15] Hutter, F., Hoos, H.H., Leyton-Brown, K., Murphy, K.P.: Time-bounded sequential parameter optimization. In: Blum, C., Battiti, R. (eds.) LION 4. LNCS, vol. 6073, pp. 281–298. Springer, Heidelberg (2010)
[16] Breiman, L.: Random forests. Machine Learning 45(1), 5–32 (2001)
[17] Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration (extended version). Technical Report TR-2010-10, UBC Computer Science (2010), http://www.cs.ubc.ca/~hutter/papers/10-TR-SMAC.pdf
[18] Bartz-Beielstein, T., Markon, S.: Tuning search algorithms for real-world applications: A regression tree based approach. In: Proc. of CEC 2004, pp. 1111–1118 (2004)
[19] Baz, M., Hunsaker, B., Brooks, P., Gosavi, A.: Automated tuning of optimization software parameters. Technical Report TR2007-7, Univ. of Pittsburgh, Industrial Engineering (2007)
[20] Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. JAIR 32, 565–606 (2008)
[21] Leyton-Brown, K., Nudelman, E., Shoham, Y.: Empirical hardness models: Methodology and a case study on combinatorial auctions. Journal of the ACM 56(4), 1–52 (2009)
[22] Hastie, T., Tibshirani, R., Friedman, J.H.: The Elements of Statistical Learning, 2nd edn. Springer Series in Statistics. Springer, Heidelberg (2009)
[23] Nell, C., Fawcett, C., Hoos, H.H., Leyton-Brown, K.: HAL: A framework for the automated analysis and design of high-performance algorithms. In: LION-5 (to appear, 2011)