Sequential Model-Based Optimization for General Algorithm Configuration (extended version)

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 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, 6]), while others also target categorical (*i.e.*, discrete-valued and unordered) domains [7, 8, 9, 10, 11]. The most prominent configuration methods are the racing algorithm F-RACE [6] and our own iterated local search algorithm PARAMILS [8, 9]. A recent competitor is the genetic algorithm GGA [10]. 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 [7, 6]. Our own group has used PARAMILS to configure highly parameterized tree search [12] and local search solvers [13] 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., [14]), 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 [15, 16, 17, 18], all published work on SMBO still has two key limitations that prevent its use for general algorithm configuration tasks: (1) it only supports numerical parameters; and (2) it only optimizes target algorithm performance for single instances.

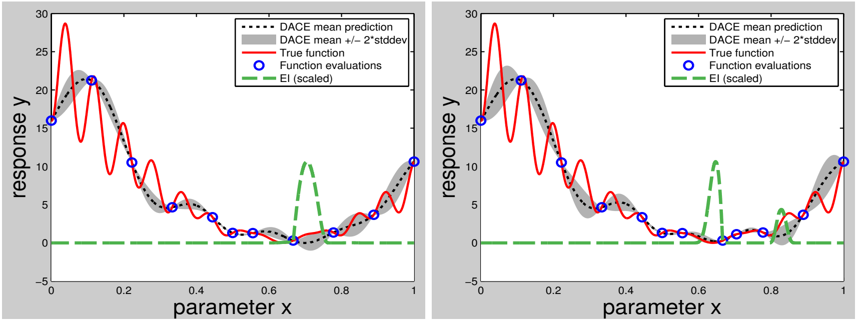
The main contribution of this paper is to remove these limitations of SMBO, and thus to make it applicable to general algorithm configuration problems with many categorical parameters and sets of benchmark instances. Specifically, we generalize four components of the SMBO framework and—based on them—define 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. In a thorough experimental analysis, we compare our new methods to the two most prominent approaches for general algorithm configuration: PARAMILS [8, 9] and GGA [10]. To demonstrate the competitive performance of our new methods, we performed configuration experiments for a wide range of 17 configuration scenarios involving a local search SAT solver, a tree search SAT solver and the commercial MIP solver CPLEX. Our results demonstrate that SMAC performed better than all previous approaches in many cases and never performed significantly worse.

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}.

Sequential model-based optimization (SMBO) iterates between building a model and gathering additional data. We illustrate a simple SMBO procedure in Figure 1. Consider a deterministic algorithm A with a single continuous parameter x and let A's runtime as a function of its parameter be described by the solid line in Figure 1(a). SMBO searches for a value of x that minimizes this runtime. Here, it is initialized by running A with the parameter values indicated by the circles in Figure 1(a). Next, SMBO fits a response surface model to the data gathered; Gaussian process (GP) models [19] are the most common choice. The black dotted line in Figure 1 represents the predictive mean of a GP model trained on the data given by the circles, and the shaded area around it quantifies the uncertainty in the predictions; this uncertainty grows with distance from the training data. SMBO uses this predictive performance model to select a promising parameter configuration for the next run of A. Promising configurations are predicted to perform well and/or lie in regions for which the model is still uncertain. These two objectives are



(a) SMBO, step 1

(b) SMBO, step 2

Fig. 1. Two steps of SMBO for the optimization of a 1D function. The true function is shown as a solid line, and the circles denote our observations. The dotted line denotes the mean prediction of a noise-free Gaussian process model (the “DACE” model), with the grey area denoting its uncertainty. Expected improvement (scaled for visualization) is shown as a dashed line.

combined in a so-called *expected improvement* (EI) criterion, which is high in regions of low predictive mean and high predictive variance (see the light dashed line in Figure 1(a); an exact formula for EI is given in Equation 3 in Section 4.3). SMBO selects a configuration with maximal EI (here, $x = 0.705$), runs A using it, and updates its model based on the result. In Figure 1(b), we show how this new data point changes the model: note the additional data point at $x = 0.705$, the greatly reduced uncertainty around it, and that the region of large EI is now split into two.

While our example captures the essence of SMBO, recent practical SMBO instantiations include more complex mechanisms for dealing with randomness in the algorithm’s performance and for reducing the computational overhead. Algorithm Framework 1 gives the general structure of the time-bounded SMBO framework we employ in this paper. It starts by running the target algorithm with some initial parameter configurations, and then iterates three steps: (1) fitting a response surface model using the existing data; (2) selecting a list of promising configurations; and (3) running the target algorithm on (some of) the selected configurations until a given time bound is reached. This time bound is related to the combined overhead, $t_{model} + t_{ei}$, due to fitting the model and selecting promising configurations.

SMBO has is 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.[14]; this is essentially the algorithm used in our simple example above. EGO is limited to optimizing continuous parameters for noise-free functions (*i.e.*, the performance of deterministic algorithms). Follow-up work in the statistics community included an approach to optimize functions across multiple environmental conditions [20] as well as the sequential kriging optimization (SKO) algorithm for handling noisy functions (*i.e.*, in our context, randomized algorithms) by Huang et al. [21]. In parallel to the latter work, Bartz-Beielstein et al. [15, 16] were the first to use the EGO approach to optimize algorithm performance. Their sequential

Algorithm Framework 1: Sequential Model-Based Optimization (SMBO)

$\mathbf{R}$ keeps track of all target algorithm runs performed so far and their performances, $\mathcal{M}$ is SMBO's model, $\vec{\mathit{\Theta}}_{new}$ 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 $\mathit{\Theta}$; instance set $\mathit{\Pi}$; cost
         metric $\hat{c}$
Output : Optimized (incumbent) parameter configuration, $\boldsymbol{\theta}_{inc}$
[$\mathbf{R}$, $\boldsymbol{\theta}_{inc}$] $\leftarrow$ Initialize($\mathit{\Theta}$, $\mathit{\Pi}$)
repeat
    [$\mathcal{M}, t_{fit}$] $\leftarrow$ FitModel($\mathbf{R}$)
    [$\vec{\mathit{\Theta}}_{new}, t_{select}$] $\leftarrow$ SelectConfigurations($\mathcal{M}$, $\boldsymbol{\theta}_{inc}$, $\mathit{\Theta}$)
    [$\mathbf{R}$, $\boldsymbol{\theta}_{inc}$] $\leftarrow$ Intensify($\vec{\mathit{\Theta}}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathcal{M}$, $\mathbf{R}$, $t_{fit} + t_{select}$, $\mathit{\Pi}$, $\hat{c}$)
until total time budget for configuration exhausted
return $\boldsymbol{\theta}_{inc}$
```

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 continuous parameters on single instances. We started our own work in SMBO by comparing SKO *vs* SPO, studying their choices for the four SMBO components [17]. We demonstrated that component *Intensify* mattered most, and improved it in our SPO$^+$ algorithm [17]. Subsequently, we showed how to reduce the 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 instead and exploit that different algorithm runs take different amounts of time. The resulting time-bounded SPO variant, TB-SPO, is the first SMBO method practical for parameter optimization given a user-specified time budget [18]. Although it was shown to significantly outperform ParamILS on some domains, it is still limited to the optimization of continuous algorithm parameters on single problem instances. In the following, we generalize the components of the time-bounded SMBO framework (of which TB-SPO is an instantiation), extending its scope to tackle general algorithm configuration problems with many categorical parameters and sets of benchmark instances.

3 Randomized Online Approximate 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

Procedure 2: Intensify($\vec{\boldsymbol{\Theta}}_{new}$, $\boldsymbol{\theta}_{inc}$, $\mathcal{M}$, $\mathbf{R}$, $t_{intensify}$, 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, $\vec{\boldsymbol{\Theta}}_{new}$; incumbent parameter setting,
         $\boldsymbol{\theta}_{inc}$; model, $\mathcal{M}$; sequence of target algorithm runs, $\mathbf{R}$; time bound, $t_{intensify}$;
         instance set, $\Pi$; cost metric, $\hat{c}$
Output : Updated sequence of target algorithm runs, $\mathbf{R}$; incumbent parameter setting, $\boldsymbol{\theta}_{inc}$
for $i := 1, \ldots, length(\vec{\boldsymbol{\Theta}}_{new})$ do
    $\boldsymbol{\theta}_{new} \leftarrow \vec{\boldsymbol{\Theta}}_{new}[i]$
    if $\mathbf{R}$ contains less than $maxR$ runs with configuration $\boldsymbol{\theta}_{inc}$ then
        $\pi \leftarrow$ instance sampled uniformly at random from $\{\pi' \in \Pi \mid \mathbf{R}$ contains less than
            or equal number of runs with $\boldsymbol{\theta}_{inc}$ for $\pi'$ than for any other $\pi'' \in \Pi\}$
        $s \leftarrow$ seed, drawn uniformly at random
        $\mathbf{R} \leftarrow$ ExecuteRun($\mathbf{R}$, $\boldsymbol{\theta}_{inc}$, $\pi$, $s$)
    $N \leftarrow 1$
    while true do
        $S_{missing} \leftarrow \langle$instance, seed$\rangle$ pairs for which $\boldsymbol{\theta}_{inc}$ was run before, but not $\boldsymbol{\theta}_{new}$
        $S_{torun} \leftarrow$random subset of $S_{missing}$ of size $\min(N, |S_{missing}|)$
        foreach $(\pi, s) \in S_{torun}$ do $\mathbf{R} \leftarrow$ ExecuteRun($\mathbf{R}$, $\boldsymbol{\theta}_{new}$, $\pi$, $s$)
        $S_{missing} \leftarrow S_{missing} \setminus S_{torun}$
        $\Pi_{common} \leftarrow$ instances for which we previously ran both $\boldsymbol{\theta}_{inc}$ and $\boldsymbol{\theta}_{new}$
        if $\hat{c}(\theta_{new}, \Pi_{common}) > \hat{c}(\theta_{inc}, \Pi_{common})$ then break
        else if $S_{missing} = \emptyset$ then $\boldsymbol{\theta}_{inc} \leftarrow \boldsymbol{\theta}_{new}$; 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 $[\mathbf{R}, \boldsymbol{\theta}_{inc}]$
```

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 instances, we also need to decide which instance to use in each run. To solve 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, $\vec{\boldsymbol{\Theta}}_{new}$, and compares each of them in turn to the current incumbent configuration. In each comparison of a new configuration, $\boldsymbol{\theta}_{new}$, to the incumbent, $\boldsymbol{\theta}_{inc}$, we first perform an additional run for the incumbent, using a randomly selected $\langle$instance, seed$\rangle$ combination, and then perform a sequence of runs using $\boldsymbol{\theta}_{new}$. For that sequence of runs, we select $\langle$instance, seed$\rangle$ combinations uniformly at random from those on which the incumbent has already run. Similar to the FOCUSEDILS algorithm [8, 9], $\boldsymbol{\theta}_{inc}$ and $\boldsymbol{\theta}_{new}$ are always compared using only instances on which they have both been run. The key difference between Procedure 2 and FOCUSEDILS's mechanism is that every comparison is based on a *different* randomly selected subset of instances and seeds; in contrast, FOCUSEDILS can be very sensitive to its fixed ordering.

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).[1] 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). Finally, we prove a convergence result for ROAR and SMAC (Section 4.4).

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, in order to handle *categorical* parameters, we adapt the most prominent previously used model class (Gaussian stochastic process models) and introduce the model class of random forests to SMBO.

[1] We previously considered random sampling approaches based on less powerful intensification mechanisms; see, *e.g.*, RANDOM* defined in [18].

A Weighted Hamming Distance Kernel Function for GP Models. Most recent work on sequential model-based optimization [14, 15, 17] uses Gaussian stochastic process models (GPs; see [19]). GP models rely on a parameterized kernel function $k : \boldsymbol{\Theta} \times \boldsymbol{\Theta} \mapsto \mathbb{R}^+$ that specifies the similarity between two parameter configurations. Previous SMBO approaches for numerical parameters typically choose the GP kernel function

$$k(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) = \exp\left[\sum_{l=1}^{d}(-\lambda_l \cdot (\boldsymbol{\theta}_{i,l} - \boldsymbol{\theta}_{j,l})^2)\right], \qquad (1)$$

where $\lambda_1, \ldots, \lambda_d$ are the kernel parameters.

For categorical parameters, we define a new, similar kernel. Instead of measuring the (weighted) squared distance, it computes a (weighted) Hamming distance:

$$k_{cat}(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) = \exp\left[\sum_{l=1}^{d}(-\lambda_l \cdot [1 - \delta(\theta_{i,l}, \theta_{j,l})])\right], \qquad (2)$$

where δ is the Kronecker delta function (ones if its two arguments are identical and zero otherwise).

For a combination of continuous parameters $\mathcal{P}_{cont}$ and categorical parameters $\mathcal{P}_{cat}$, we apply the combined kernel

$$K_{mixed}(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) = \exp\left[\sum_{l \in \mathcal{P}_{cont}} (-\lambda_l \cdot (\theta_{i,l} - \theta_{j,l})^2) + \sum_{l \in \mathcal{P}_{cat}} (-\lambda_l \cdot [1 - \delta(\theta_{i,l}, \theta_{j,l})])\right].$$

Although K_{mixed} is a straightforward generalization of the standard Gaussian kernel in Equation 1, we are not aware of any prior use of this kernel or proof that it is indeed a valid kernel function.[2] We provide this proof in the appendix. Since Gaussian stochastic processes are kernel-based learning methods and since K_{mixed} is a valid kernel function, it can be swapped in for the Gaussian kernel without changing any other component of the GP model. Here, we use the same projected process (PP) approximation of GP models [19] as in TB-SPO [18].

Random Forests. The new default model we use in SMAC is based on random forests [23], a standard machine learning tool for regression and classification. 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 (both in terms of runtime and solution quality) of heuristic algorithms (e.g., [24, 25]). Random forests share this benefit and typically yield more accurate predictions [23]; 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

[2] Couto [22] gives a recursive kernel function for categorical data that is related since it is also based on a Hamming distance.

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.[3] 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 $\boldsymbol{\theta}$. Usually, the tree prediction for a parameter configuration $\boldsymbol{\theta}_{n+1}$ is the mean of the data points in the leaf one ends up in when propagating $\boldsymbol{\theta}_{n+1}$ down the tree. We adapted this mechanism to instead predict the user-defined cost metric of that data, *e.g.*, the median of the data points in that leaf.

Transformations of the Cost Metric. 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 [26] 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. For example, take a simple case where there is only one parameter configuration $\boldsymbol{\theta}$ for which we measured runtimes $(r_1, \ldots, r_{10})$=$(2^1, 2^2, \ldots, 2^{10})$. While the true arithmetic mean of these runs is roughly 205, a GP model trained on this data using a log transformation would predict the mean to be $\exp\left(\text{mean}((\log(r_1), \ldots, \log(r_{10}))\right) \approx 45$. This is because the arithmetic mean of the logs is the log of the *geometric* mean:

$$\text{geometric mean} = \sqrt[n]{\prod_{i=1}^{n} x_i} = \left[\exp\left(\sum_{i=1}^{n} \log(x_i)\right)\right]^{(1/n)}$$

$$= \exp\left[\frac{1}{n}\sum_{i=1}^{n} \log(x_i)\right] = \exp\left(\text{mean of logs}\right).$$

For GPs, it is not clear how to fix this problem. 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

[3] 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.

for N: small N leads to poor generalization to test data, while large N leads to a prohibitive N-fold slowdown in the cost of each evaluation. (This is the same problem faced by the PARAMILS instantiation BASICILS(N) [8].) 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 [27] 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 [28, 26].

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. All features are listed in Figure 2. For MIP instances we computed 39 features, including features based on graph representations, an LP relaxation, the objective function, and the linear constraint matrix. All features are listed in Figure 3.

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, such as SATzilla [28, 26], 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. As a corollary, the time required for feature computation is not as crucial in algorithm configuration as it is in per-instance approaches: in per-instance approaches, feature computation has to be counted as part of the time required to solve test instances, while in algorithm configuration no features are computed for test instances at all. In fact, features for the training instances may well be the result of an extensive offline analysis of those training instances, or can even be taken from the literature. Computing the features we used here took an average of 30 seconds for the SAT domains, and 4.4 seconds for the MIP domains.

Learning a random forest (RF) model can be slow in the presence of many features. To speed it up, we applied *principal component analysis* (see, *e.g.*, [31]), to project the feature matrix into a lower-dimensional subspace spanned by the seven orthogonal vectors along which it has maximal variance.

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 $\{([\boldsymbol{\theta}_1, \boldsymbol{x}_1], o_1), \dots, ([\boldsymbol{\theta}_n, \boldsymbol{x}_n], o_n)\}$. From this data, we wish to learn a model that takes

Problem Size Features:

1.–2. **Number of variables and clauses in original formula**: denoted v and c, respectively

3.–4. **Number of variables and clauses after simplification with SATelite**: denoted v' and c', respectively

5.–6. **Reduction of variables and clauses by simplification**: (v-v')/v' and (c-c')/c'

7. **Ratio of variables to clauses**: v'/c'

Variable-Clause Graph Features:

8.–12. **Variable node degree statistics:** mean, variation coefficient, min, max, and entropy

13.–17. **Clause node degree statistics:** mean, variation coefficient, min, max, and entropy

Variable Graph Features:

18–21. **Node degree statistics:** mean, variation coefficient, min, and max

22.–26. **Diameter:** mean, variation coefficient, min, max, and entropy

27.–31. **Clustering Coefficient:** mean, variation coefficient, min, max, and entropy

Clause Graph Features:

32–36. **Node degree statistics:** mean, variation coefficient, min, max, and entropy

Balance Features:

37.–41. **Ratio of positive to negative literals in each clause:** mean, variation coefficient, min, max, and entropy

42.–46. **Ratio of positive to negative occurrences of each variable:** mean, variation coefficient, min, max, and entropy

47.–49. **Fraction of unary, binary, and ternary clauses**

Proximity to Horn Formula:

50. **Fraction of Horn clauses**

51.–55. **Number of occurrences in a Horn clause for each variable:** mean, variation coefficient, min, max, and entropy

DPLL Probing Features:

56.–60. **Number of unit propagations:** computed at depths 1, 4, 16, 64 and 256

61.–62. **Search space size estimate:** mean depth to contradiction, estimate of the log of number of nodes

LP-Based Features:

63.–66. **Integer slack vector:** mean, variation coefficient, min, and max

67. **Ratio of integer variables in LP solution**

68. **Objective function value of LP solution**

Local Search Probing Features, based on 2 seconds of running each of SAPS and GSAT:

69.–78. **Number of steps to the best local minimum in a run:** mean, median, variation coefficient, 10th and 90th percentiles

79.–82. **Average improvement to best in a run:** mean and coefficient of variation of improvement per step to best solution

83.–86. **Fraction of improvement due to first local minimum:** mean and variation coefficient

87.–90. **Coefficient of variation of the number of unsatisfied clauses in each local minimum:** mean and variation coefficient

Clause Learning Features (based on 2 seconds of running Zchaff_rand):

91.–99. **Number of learned clauses:** mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

100.–108. **Length of learned clauses:** mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

Survey Propagation Features

109.–117. **Confidence of survey propagation:** For each variable, compute the higher of $P(true)/P(false)$ or $P(false)/P(true)$. Then compute statistics across variables: mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

118.–126. **Unconstrained variables:** For each variable, compute $P(unconstrained)$. Then compute statistics across variables: mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

Fig. 2. 11 groups of SAT features; these were introduced in [28, 26, 29].

as input a parameter configuration $\boldsymbol{\theta}$ and predicts performance across all training instances. For GP models, there exists an approach from the statistics literature to predict mean performance across problem instances [20]. However, due to the issue discussed in Section 4.1, when using log transformations this approach would not model the cost metric the user specifies; *e.g.*, instead of the arithmetic mean it would model geometric

Problem Size Features:

1.–2. **Number of variables and constraints**: denoted n and m, respectively
3. **Number of nonzero entries in the linear constraint matrix, $\boldsymbol{A}$**

Variable-Constraint Graph Features:

4–7. **Variable node degree statistics**: mean, max, min, and stddev
8–11. **Constraint node degree statistics**: mean, max, min, and stddev

Variable Graph (VG) Features:

12–17. **Node degree statistics**: max, min, stddev, 25% and 75% quantiles
18–19. **Clustering Coefficient:** mean and stddev
20. **Edge Density**: number of edges in the VG divided by the number of edges in a complete graph having the same number of nodes

LP-Based Features:

21–23. **Integer slack vector:** mean, max, L_2 norm
24. **Objective function value of LP solution**

Objective Function Features:

25. **Standard deviation of normalized coefficients:** $\{c_i/m\}_{i=1}^n$
26. **Standard deviation of** $\{c_i/n_i\}_{i=1}^n$, where n_i denotes the number of nonzero entries in column i of $\boldsymbol{A}$
27. **Standard deviation of** $\{c_i/\sqrt{n_i}\}_{i=1}^n$

Linear Constraint Matrix Features:

28.–29. **Distribution of normalized constraint matrix entries, $\boldsymbol{A}_{i,j}/b_i$**: mean and stddev (only of elements where $b_i \neq 0$)
30.–31. **Variation coefficient of normalized absolute nonzero entries per row**: mean and stddev

Variable Type Features:

32.–33. **Support size of discrete variables**: mean and stddev
34. **Percent unbounded discrete variables**
35. **Percent continuous variables**

General Problem Type Features:

36. **Problem type**: categorical feature attributed by CPLEX (LP, MILP, FIXEDMILP, QP, MIQP, FIXEDMIQP, MIQP, QCP, or MIQCP)
37. **Number of quadratic constraints**
38. **Number of nonzero entries in matrix of quadratic coefficients of objective function, $\boldsymbol{Q}$**
39. **Number of variables with nonzero entries in $\boldsymbol{Q}$**

Fig. 3. Eight groups of features for the mixed integer programming problem. These general MIP features have been introduced in [11] as a generalization of features for the combinatorial winner determination problem in [30].

mean. This problem would be particularly serious in the case of multiple instances, as performance often varies by orders of magnitude from one instance to another. As a consequence, we did not implement a version of SMAC(PP) for multiple instances at this time. Instead, we adapted RF models to handle predictions across multiple instances. All input dimensions are handled equally when constructing the random forest, 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 parameter configuration and each instance; next, we combine these predictions with the user-defined cost metric (*e.g.*, arithmetic mean runtime); 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* *($EI(\boldsymbol{\theta})$)* [14] 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 [17] 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

$$EI(\boldsymbol{\theta}) := E[I_{\text{exp}}(\boldsymbol{\theta})] = f_{min}\Phi(v) - e^{\frac{1}{2}\sigma^2_{\boldsymbol{\theta}}+\mu_\theta} \cdot \Phi(v - \sigma_{\boldsymbol{\theta}}), \qquad (3)$$

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 $\boldsymbol{\theta}_{inc}$.[4]

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 [15, 16, 17, 18] 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.[5] This search is similar in spirit to PARAMILS [8, 9], but instead of algorithm performance it optimizes $EI(\boldsymbol{\theta})$ (see Equation 3), 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.)

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

[4] In TB-SPO [18], 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.

[5] 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.

models. More precisely, we alternate between configurations from the list and additional configurations sampled uniformly at random.

4.4 Theoretical Analysis of SMAC and ROAR

In this section, we provide a convergence proof for SMAC (and ROAR) for finite configuration spaces. 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. The proof is straight-forward, following the same arguments as a previous proof about FocusedILS (see [9]). Nevertheless, we give it here for completeness.

Definition 1 (Consistent estimator) $\hat{c}_N(\boldsymbol{\theta})$ *is a* consistent estimator *for* $c(\boldsymbol{\theta})$ *iff*

$$\forall \epsilon > 0 \, : \, \lim_{N \to \infty} P(|\hat{c}_N(\boldsymbol{\theta}) - c(\boldsymbol{\theta})| < \epsilon) = 1.$$

When we estimate that a parameter configuration's true cost $c(\boldsymbol{\theta})$ based on N runs is $\hat{c}_N(\boldsymbol{\theta})$, and when $\hat{c}_N(\boldsymbol{\theta})$ is a consistent estimator of $c(\boldsymbol{\theta})$, cost estimates become increasingly reliable as N approaches infinity, eventually eliminating the possibility of mistakes in comparing two parameter configurations. The following lemma is exactly Lemma 8 in [9]; for the proof, see that paper.

Lemma 2 (No mistakes for $N \to \infty$) *Let $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2 \in \boldsymbol{\Theta}$ be any two parameter configurations with $c(\boldsymbol{\theta}_1) < c(\boldsymbol{\theta}_2)$. Then, for consistent estimators $\hat{c}_N$, $\lim_{N\to\infty} P(\hat{c}_N(\boldsymbol{\theta}_1) \geq \hat{c}_N(\boldsymbol{\theta}_2)) = 0$.*

All that remains to be shown is that SMAC evaluates each parameter configuration an unbounded number of times.

Lemma 3 (Unbounded number of evaluations) *Let $N(J, \boldsymbol{\theta})$ denote the number of runs* SMAC *has performed with parameter configuration $\boldsymbol{\theta}$ at the end of SMBO iteration J. Then, if SMBO's parameter $maxR$ is set to ∞, for any constant K and configuration $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ (with finite $\boldsymbol{\Theta}$), $\lim_{J\to\infty} P[N(J, \boldsymbol{\theta}) \geq K] = 1$.*

Proof. In each SMBO iteration, SMAC *evaluates at least one random configuration (performing at least one new run for it since $maxR = \infty$), and with a probability of $p = 1/|\boldsymbol{\Theta}|$, this is configuration $\boldsymbol{\theta}$. Hence, the number of runs performed with $\boldsymbol{\theta}$ is lower-bounded by a binomial random variable $\mathcal{B}(k; J, p)$. Then, for any constant $k < K$ we obtain $\lim_{J\to\infty} \mathcal{B}(k; J, p)$ Thus, $\lim_{J\to\infty} P[N(J, \boldsymbol{\theta}) \geq K] = 1$.*

Theorem 4 (Convergence of SMAC) *When* SMAC *with $maxR = \infty$ optimizes a cost measure c based on a consistent estimator $\hat{c}_N$ and a finite configuration space $\boldsymbol{\Theta}$, the probability that it finds the true optimal parameter configuration $\boldsymbol{\theta}^* \in \boldsymbol{\Theta}$ approaches one as the time allowed for configuration goes to infinity.*

Proof. *Each SMBO iteration takes finite time. Thus, as time goes to infinity so does the number of SMBO iterations, J. According to Lemma 3, as J goes to infinity $N(\boldsymbol{\theta})$ grows unboundedly for each $\boldsymbol{\theta} \in \boldsymbol{\Theta}$. For each $\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$, as $N(\boldsymbol{\theta}_1)$ and $N(\boldsymbol{\theta}_2)$ go to infinity, Lemma 2 states that in a pairwise comparison, the truly better configuration will be preferred. Thus eventually,* SMAC *visits all finitely many parameter configurations and prefers the best one over all others with probability arbitrarily close to one.*

We note that this convergence result holds regardless of the model type used. In fact, it even holds for a simple round robin procedure that loops through parameter configurations. We thus rely on empirical results to assess SMAC. We present these in the following sections, after explaining how to build the relevant models.

5 Experimental Evaluation

We now compare the performance of SMAC, ROAR, TB-SPO [18], GGA [10], and PARAMILS (in particular, FOCUSEDILS 2.3) [9] for a wide variety of configuration scenarios, aiming to target algorithm runtime for solving SAT and MIP problems. 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 used 17 configuration scenarios from the literature, involving the configuration of the local search SAT solver SAPS [32] (4 parameters), the tree search solver SPEAR [33] (26 parameters), and the most widely used commercial mixed integer programming solver, IBM ILOG CPLEX[6] (76 parameters). SAPS is a dynamic local search algorithm, and its four continuous parameters control the scaling and smoothing of clause weights, as well as the percentage of random steps. We use its UBCSAT implementation [34]. SPEAR is a tree search algorithm for SAT solving developed for industrial instances, and with appropriate parameter settings it is the best available solver for certain types of SAT-encoded hardware and software verification instances [12]. SPEAR has 26 parameters, including ten categorical, four integer, and twelve continuous parameters. The categorical parameters mainly control heuristics for variable and value selection, clause sorting, resolution ordering, and also enable or disable optimizations, such as the pure literal rule. The continuous and integer parameters mainly deal with activity, decay, and elimination of variables and clauses, as well as with the interval of randomized restarts and percentage of random choices. CPLEX is the most-widely used commercial optimization tool for solving MIPs, currently used by over 1 300 corporations and government agencies, along with researchers at over 1 000 universities. In defining CPLEX's configuration space, we were careful to keep all parameters fixed that change the problem formulation (*e.g.*, parameters such as the optimality gap below which a solution is considered optimal). The 76 parameters we selected affect all aspects of CPLEX. They include 12 preprocessing parameters (mostly

[6] http://ibm.com/software/integration/optimization/cplex-optimizer

categorical); 17 MIP strategy parameters (mostly categorical); 11 categorical parameters deciding how aggressively to use which types of cuts; 9 numerical MIP "limits" parameters; 10 simplex parameters (half of them categorical); 6 barrier optimization parameters (mostly categorical); and 11 further parameters. Most parameters have an "automatic" option as one of their values. We allowed this value, but also included other values (all other values for categorical parameters, and a range of values for numerical parameters).

In all 17 configuration scenarios, we terminated target algorithm runs at $\kappa_{max} = 5$ seconds, the same per-run cutoff time used in previous work for these scenarios. 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}). Using PAR-10 to compare the remaining configuration procedures, we found that our qualitative results did not change. All instances we used are available at http://www.cs.ubc.ca/labs/beta/Projects/AAC.

Parameter transformations. Some numerical parameters naturally vary on a non-uniform scale (*e.g.*, a parameter θ with an interval $[100, 1600]$ 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. For GGA, we used parameter settings recommended by its 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 [10] of GGA and FOCUSEDILS, capping was disabled in FOCUSEDILS; this explains its poor performance there and its better performance here.

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 [18], which aim to minimize SAPS's runtime on 6 single SAT-encoded instances, 3 each from quasigroup completion (QCP [35]) and from small world graph colouring (SWGCP [36]). We also used 5 similar new configuration scenarios, which aim to optimize SPEAR for 5 further SAT-encoded instances: 3 from software verification (SWV [37]) and 2 from IBM bounded model checking (IBM [38]; we only used 2 low quantiles of this hard distribution since SPEAR could not solve the instance at the 75% quantile within the cutoff time). The configuration scenarios are named `algorithm-distribution-quantile`: ***e.g.***, SAPS-QCP-MED aims to optimize SAPS performance on a median-hard QCP instance. The time budget for each algorithm configuration run was 30 CPU minutes, exactly following [18].

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(PP) and TB-SPO are very similar, and both performed best.[7] While TB-SPO does not apply in the remaining configuration scenarios, our more general SMAC method achieved the best performance in all of them. For all-numerical parameters, SMAC performed slightly better using PP models, while in the presence of categorical parameters the RF models performed better. 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 4 visualizes each configurator's 25 test performances for all scenarios. We note that SMAC 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

[7] In this special case, the two only differ in the expected improvement criterion and its optimization.

Scenario	Unit	Full configuration space					Discretized configuration space				
		SMAC	SMAC(PP)	TB-SPO	ROAR	GGA	SMAC	SMAC(PP)	ROAR	F-ILS	GGA
SAPS-QCP-MED	$[\cdot 10^{-2} s]$	4.70	**4.54**	**4.58**	4.72	6.28	**5.27**	**5.27**	**5.25**	5.50	6.24
SAPS-QCP-Q075	$[\cdot 10^{-1} s]$	**2.29**	**2.21**	**2.22**	2.34	2.74	**2.87**	**2.93**	**2.92**	**2.91**	**2.98**
SAPS-QCP-Q095	$[\cdot 10^{-1} s]$	**1.37**	**1.35**	**1.35**	1.55	1.75	**1.51**	**1.58**	**1.57**	**1.57**	1.95
SAPS-SWGCP-MED	$[\cdot 10^{-1} s]$	**1.61**	**1.65**	**1.63**	**1.7**	2.48	**2.54**	2.59	**2.58**	2.57	2.71
SAPS-SWGCP-Q075	$[\cdot 10^{-1} s]$	**2.11**	**2.2**	**2.48**	2.32	3.19	**3.26**	3.41	3.38	3.55	3.55
SAPS-SWGCP-Q095	$[\cdot 10^{-1} s]$	**2.36**	**2.56**	**2.69**	**2.49**	**3.13**	**3.65**	3.78	**3.79**	**3.75**	**3.77**
SPEAR-IBM-Q025	$[\cdot 10^{-1} s]$	**6.24**	6.31	—	6.31	6.33	**6.21**	6.27	6.30	6.31	6.30
SPEAR-IBM-MED	$[\cdot 10^{0} s]$	3.28	**3.07**	—	3.36	3.35	**3.16**	**3.18**	3.38	3.47	3.84
SPEAR-SWV-MED	$[\cdot 10^{-1} s]$	**6.04**	6.12	—	6.11	6.14	**6.05**	6.12	6.14	6.11	6.15
SPEAR-SWV-Q075	$[\cdot 10^{-1} s]$	**5.76**	5.83	—	5.88	5.83	**5.76**	5.89	5.89	5.88	5.84
SPEAR-SWV-Q095	$[\cdot 10^{-1} s]$	**8.38**	8.5	—	8.55	8.47	**8.42**	8.51	8.53	8.58	8.49

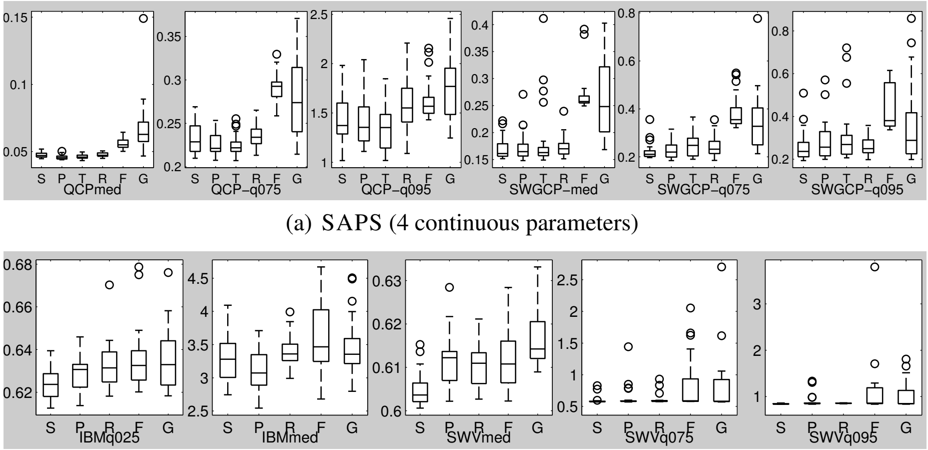
Table 1. Comparison of algorithm configuration procedures for optimizing parameters on single problem instances. We performed 25 independent runs of each configuration procedure and report the median of the 25 test performances (mean runtimes across 1 000 target algorithm runs with the found configurations). 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.

SAPS with PARAMILS in previous work [8, 9]) 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 [9] to evaluate PARAMILS's performance, plus one further CPLEX scenario, and we used the same time budget of 5 hours per configuration run. These instances come from the following domains: quasigroup completion, QCP [35]; small world graph colouring, SWGCP [36]; winner determination in combinatorial auctions, REGIONS100 [39]; mixed integer knapsack, MIK [40].

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. GGA performed slightly better for optimizing CPLEX than ROAR, but also significantly worse than either



(a) SAPS (4 continuous parameters)

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

Fig. 4. 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, 'P' for SMAC(PP), 'T' for TB-SPO, 'R' for ROAR, 'F' for FocusedILS, and 'G' for GGA.

FocusedILS or SMAC. Figure 5 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 [9]; 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.

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

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

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, based on a Mann-Whitney U test.

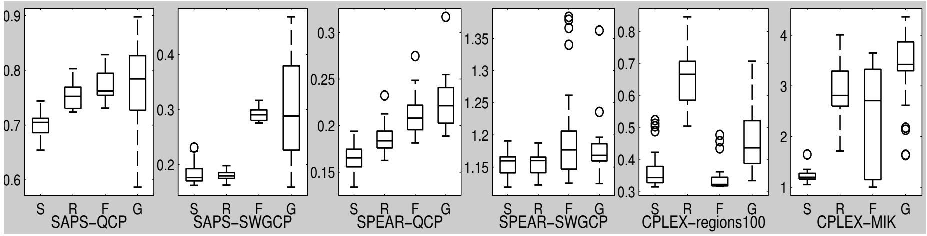


Fig. 5. Visual comparison of configuration procedures for general algorithm configuration scenarios. For each configurator and scenario, we show boxplots for the runtime data underlying Table 2, for the full configuration space (discretized for FOCUSEDILS). 'S' stands for SMAC, 'R' for ROAR, 'F' for FOCUSEDILS, and 'G' for GGA. FOCUSEDILS does not apply for the full configuration space, denoted by "—".

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, demonstrating the strength of our methods. 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 further, in particular by extending its models to handle partly censored data (stemming from prematurely terminated runs). We also plan to use our response surface models to characterize the importance of individual parameters and their interactions, and to study interactions between parameters and instance features. Finally, we plan to use our response surface models to determine good configurations on a per-instance basis.

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. FH gratefully acknowledges support from a postdoctoral research fellowship by the Canadian Bureau for International Education. HH and KLB gratefully acknowledge support from NSERC through their respective discovery grants, and from the MITACS NCE through a seed project grant.

A Proof of Theorem 1

In order to prove the validity of kernel function K_{mixed}, we first prove the following Lemma:

Lemma 5 (Validity of K_{ham}) *For any finite domain Θ, the kernel function $K_{ham} : \Theta \times \Theta \rightarrow \mathbb{R}$ defined as*

$$K_{ham}(x, z) = \exp(-\lambda \cdot [1 - \delta(x, z)]) \tag{4}$$

is a valid kernel function.

Proof. We use the facts that any constant is a valid kernel function, and that the space of kernel functions is closed under addition and multiplication. We also use the fact that a kernel function $k(x, z)$ is valid if we can find an embedding ϕ such that $k(x, z) = \phi(x)^{\intercal} \cdot \phi(z)$ [41].

Let $a_1, \ldots, a_m$ denote the finitely many elements of Θ_i, and for each element a_i define an m-dimensional indicator vector $\boldsymbol{v}_{a_i}$ that contains only zeros except at position i, where it is one. Define a kernel function $k_1(x, z)$ for $x, z \in \Theta_i$ as the dot-product of embeddings $\phi(x)$ and $\phi(z)$ in an m-dimensional space:

$$k_1(x, z) = \boldsymbol{v}_x^{\intercal} \cdot \boldsymbol{v}_z = \sum_{i=1}^{m} \boldsymbol{v}_x(i) \cdot \boldsymbol{v}_z(i) = 1 - \delta(x, z).$$

To bring this in the form of Equation 4, we add the constant kernel function

$$k_2(x, z) = c = \frac{\exp(-\lambda)}{1 - \exp(-\lambda)},$$

and then multiply by the constant kernel function

$$k_3(x, z) = 1/(1 + c) = 1 - \exp(-\lambda).$$

We can thus rewrite function K_{ham} as the product of valid kernels, thereby proving its validity:

$$\begin{aligned} K_{ham}(x, z) &= (k_1(x, z) + k_2(x, z)) \cdot k_3(x, z) \\ &= \begin{cases} 1 & \textit{if } x = z \\ \exp(-\lambda) & \textit{otherwise} \end{cases} \\ &= \exp\left[-\lambda \delta(x \neq z)\right]. \end{aligned}$$

Theorem 1 (Validity of K_{mixed}). *The kernel $K_{mixed} : \boldsymbol{\Theta} \times \boldsymbol{\Theta} \to \mathbb{R}$ defined as*

$$K_{mixed}(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) = \exp\left[\sum_{l \in \mathcal{P}_{cont}} (-\lambda_l \cdot (\theta_{i,l} - \theta_{j,l})^2) + \sum_{l \in \mathcal{P}_{cat}} (-\lambda_l \cdot [1 - \delta(\theta_{i,l}, \theta_{j,l})])\right].$$

is a valid kernel function for arbitrary configuration spaces.

Proof. *Since K_{mixed} is a product of the valid Gaussian kernel*

$$K(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) = \exp\left[\sum_{l \in \mathcal{P}_{cont}} (-\lambda_l \cdot (\theta_{i,l} - \theta_{j,l})^2)\right]$$

and one K_{ham} kernel for each categorical parameter, it is a valid kernel.

References

[1] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Proc. of CPAIOR-10*, pages 186–202, 2010.

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

[3] J. Gratch and G. Dejong. Composer: A probabilistic solution to the utility problem in speed-up learning. In *Proc. of AAAI-92*, pages 235–240, 1992.

[4] S. P. Coy, B. L. Golden, G. C. Runger, and E. A. Wasil. Using experimental design to find effective parameter settings for heuristics. *Journal of Heuristics*, 7(1):77–97, 2001.

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

[6] P. Balaprakash, M. Birattari, and T. Stützle. Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement. In *Proc. of MH-07*, pages 108–122, 2007.

[7] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proc. of GECCO-02*, pages 11–18, 2002.

[8] F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proc. of AAAI-07*, pages 1152–1157, 2007.

[9] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *JAIR*, 36:267–306, October 2009.

[10] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proc. of CP-09*, pages 142–157, 2009.

[11] F. Hutter. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, University Of British Columbia, Vancouver, Canada, 2009.

[12] F. Hutter, D. Babić, H. H. Hoos, and A. J. Hu. Boosting Verification by Automatic Tuning of Decision Procedures. In *Proc. of FMCAD'07*, pages 27–34, 2007.

[13] A. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. In *Proc. of IJCAI-09*, pages 517–524, 2009.

[14] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black box functions. *Journal of Global Optimization*, 13:455–492, 1998.

[15] T. Bartz-Beielstein, C. Lasarczyk, and M. Preuss. Sequential parameter optimization. In *Proc. of CEC-05*, pages 773–780. IEEE Press, 2005.

[16] T. Bartz-Beielstein. *Experimental Research in Evolutionary Computation: The New Experimentalism*. Natural Computing Series. Springer Verlag, Berlin, 2006.

[17] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. P. Murphy. An experimental investigation of model-based parameter optimisation: SPO and beyond. In *Proc. of GECCO-09*, pages 271–278, 2009.

[18] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. P. Murphy. Time-bounded sequential parameter optimization. In *Proc. of LION-4*, pages 281–298, 2010.

[19] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[20] B. J. Williams, T. J. Santner, and W. I. Notz. Sequential design of computer experiments to minimize integrated response functions. *Statistica Sinica*, 10:1133–1152, 2000.

[21] D. Huang, T. T. Allen, W. I. Notz, and N. Zeng. Global optimization of stochastic black-box systems via sequential kriging meta-models. *Journal of Global Optimization*, 34(3):441–466, 2006.

[22] Julia Couto. Kernel k-means for categorical data. In *Advances in Intelligent Data Analysis VI (IDA-05)*, volume 3646 of *LNCS*, pages 46–56, 2005.

[23] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[24] T. Bartz-Beielstein and S. Markon. Tuning search algorithms for real-world applications: A regression tree based approach. In *Proc. of CEC-04*, pages 1111–1118, 2004.

[25] M. Baz, B. Hunsaker, P. Brooks, and A. Gosavi. Automated tuning of optimization software parameters. Technical Report TR2007-7, University of Pittsburgh, Department of Industrial Engineering, 2007.

[26] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *JAIR*, 32:565–606, June 2008.

[27] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):1–52, 2009.

[28] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Proc. of CP-04*, 2004.

[29] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla2009: an automatic algorithm portfolio for sat. Solver description, SAT competition 2009, 2009.

[30] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Proc. of CP-02*, pages 556–572, 2002.

[31] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer Verlag, 2nd edition, 2009.

[32] F. Hutter, D. A. D. Tompkins, and H. H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proc. of CP-02*, pages 233–248, 2002.

[33] D. Babić. *Exploiting Structure for Scalable Software Verification*. PhD thesis, University of British Columbia, Vancouver, Canada, 2008.

[34] D. A. D. Tompkins and H. H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT & MAX-SAT. In *Proc. of SAT-04*, pages 306–320, 2004.

[35] C. P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proc. of AAAI-97*, pages 221–226, 1997.

[36] I. P. Gent, H. H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. In *Proc. of AAAI-99*, pages 654–660, 1999.

[37] D. Babić and A. J. Hu. Structural Abstraction of Software Verification Conditions. In *Proc. of CAV-07*, pages 366–378, 2007.

[38] E. Zarpas. Benchmarking SAT Solvers for Bounded Model Checking. In *Proc. of SAT-05*, pages 340–354, 2005.

[39] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proc. of EC'00*, pages 66–76, New York, NY, USA, 2000. ACM.

[40] A. Atamtürk. On the facets of the mixed–integer knapsack polyhedron. *Mathematical Programming*, 98:145–175, 2003.

[41] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA, 2004.