Chapter 3
Automated Algorithm Configuration and Parameter Tuning

Holger H. Hoos

3.1 Introduction

Computationally challenging problems arise in the context of many applications, and the ability to solve these as efficiently as possible is of great practical, and often also economic, importance. Examples of such problems include scheduling, time-tabling, resource allocation, production planning and optimisation, computer-aided design and software verification. Many of these problems are $\mathcal{NP}$-hard and considered computationally intractable, because there is no polynomial-time algorithm that can find solutions in the worst case (unless $\mathcal{NP}=\mathcal{P}$). However, by using carefully crafted heuristic techniques, it is often possible to solve practically relevant instances of these 'intractable' problems surprisingly effectively (see, e.g., 55, 3, 54) [1].

The practically observed efficacy of these heuristic mechanisms remains typically inaccessible to the analytical techniques used for proving theoretical complexity results, and therefore needs to be established empirically, on the basis of carefully designed computational experiments. In many cases, state-of-the-art performance is achieved using several heuristic mechanisms that interact in complex, non-intuitive ways. For example, a DPLL-style complete solver for SAT (a prototypical $\mathcal{NP}$-complete problem with important applications in the design of reliable soft- and hardware) may use different heuristics for selecting variables to be instantiated and the values first explored for these variables, as well as heuristic mechanisms for managing and using logical constraints derived from failed solution attempts. The activation, interaction and precise behaviour of those mechanisms is often controlled by parameters, and the settings of such parameters have a substantial impact on the

Holger H. Hoos
Department of Computer Science, University of British Columbia, 2366 Main Mall, Vancouver, BC, V6T 1Z4, Canada, e-mail: hoos@cs.ubc.ca

[1] We note that the use of heuristic techniques does not imply that the resulting algorithms are necessarily incomplete or do not have provable performance guarantees, but often results in empirical performance far better than the bounds guaranteed by rigorous theoretical analysis.

Y. Hamadi et al. (eds.), *Autonomous Search*,
DOI 10.1007/978-3-642-21434-9_3,

efficacy with which a heuristic algorithm solves a given problem instance or class of problem instances. For example, the run-time of CPLEX 12.1 – a widely used, commercial solver for mixed integer programming problems – has recently been demonstrated to vary by up to a factor of over 50 with the settings of 76 user-accessible parameters [42].

A problem routinely encountered by designers as well as end users of parameterised algorithms is that of finding parameter settings (or *configurations)* for which the empirical performance on a given set of problem instances is optimised. Formally, this *algorithm configuration* or *parameter tuning* problem can be stated as follows:

Given
- an algorithm A with parameters $p_1, \ldots, p_k$ that affect its behaviour,
- a space C of configurations (i.e., parameter settings), where each configuration $c \in C$ specifies values for A's parameters such that A's behaviour on a given problem instance is completely specified (up to possible randomisation of A),
- a set of problem instances I,
- a performance metric m that measures the performance of A on instance set I for a given configuration c,

find a configuration $c^* \in C$ that results in optimal performance of A on I according to metric m.

In the context of this problem, the algorithm whose performance is to be optimised is often called the *target algorithm*, and we use $A(c)$ to denote target algorithm A under a specific configuration c. The set of values any given parameter p can take is called the *domain* of p. Depending on the given target algorithm, various types of parameters may occur. *Categorical parameters* have a finite, unordered set of discrete values; they are often used to select from a number of alternative mechanisms or components. Using *Boolean parameters*, heuristic mechanisms can be activated or deactived, while the behaviour and interaction of these mechanisms is often controlled by *integer-* and *real-valued parameters* (the former of which are a special cases of *ordinal parameters*, whose domains are discrete and ordered). *Conditional parameters* are only active when other parameters are set to particular values; they routinely arise in the context of mechanisms that are activated or selected using some parameter, and whose behaviour is then controlled by other parameters (where the latter parameters conditionally depend on the former). Sometimes, it is useful to place additional constraints on configurations, e.g., to exclude certain combinations of parameter values that would lead to ill-defined, incorrect or otherwise undesirable behaviour of a given target algorithm.

Clearly, the number and types of parameters, along with the occurrence of conditional parameters and constraints on configurations, determine the nature of the configuration space C and have profound implications on the methods to be used for searching performance-optimising configurations within that space. These methods range from well-known numerical optimisation procedures, such as the Nelder-Mead Simplex algorithm [49, 13] or the more recent gradient-free CMA-ES algo-

rithm [25, 27, 26], to approaches based on experimental design methods (see, e.g., 11, 5, 1), response-surface models (see, e.g., 44, 7) and stochastic local search procedures (see, e.g., 36, 37).

In general, when configuring a specific target algorithm, it is desirable to find parameter configurations that work well on problem instances other than those in the given instance set I. To this end, care needs to be taken in selecting the instances in I to be representative of the kinds of instances to which the optimised target algorithm configuration is expected to be applied. Difficulties can arise when I is small, yet contains very different types of instances. To recognise situations in which a configured target algorithm, $A(c^*)$, fails to perform well when applied to instances other than those used in the configuration process, it is advisable to test it on a set of instances not contained in I; this can be done by including in I only part of the overall set of instances available, or by means of cross-validation.

It is also advisable to investigate performance variation of $A(c^*)$ over instance set I, since, depending on the performance metric m used for the configuration of A and differences between instances in I, the optimised configuration c^* might represent a trade-off between strong performance on some instances and weaker performance on others. In particular, when using a robust statistic, such as median run-time, as a performance metric, poor performance on large parts of a given instance set can result. To deal effectively with target algorithm runs in which no solution was produced (in particular, time-outs encountered when optimising run-time), it is often useful to use a performance metric based on penalised averaging, in which a fixed penalty is assigned to any unsuccessful run of A (see also 37).

In the existing literature, the terms *algorithm configuration* and *parameter tuning* are often used interchangeably. We prefer to use *parameter tuning* in the context of target algorithms with relatively few parameters with mostly real-valued domains, and *algorithm configuration* in the context of target algorithms with many categorical parameters. Following Hoos [28], we note that algorithm configuration problems arise when dealing with an algorithm schema that contains a number of instantiable components (typically, subprocedures or functions), along with a discrete set of concrete choices for each of these. While most standard numerical optimisation methods are not applicable to these types of algorithm configuration problems, F-Race [11, 5], Calibra [1] and ParamILS [36, 37] have been used successfully in this context. However, so far only ParamILS has been demonstrated to be able to deal with the vast design spaces resulting from schemata with many independently instantiable components (see, e.g., 45, 66), and promising results have been achieved by a genetic programming procedure applied to the configuration of local search algorithms for SAT [18, 19], as well as by a recent gender-based genetic algorithm [2].

In the remainder of this chapter, we discuss three classes of methods for solving algorithm configuration and parameter tuning problems. *Racing procedures* iteratively evaluate target algorithm configurations on problem instances from a given set and use statistical hypothesis tests to eliminate candidate configurations that are significantly outperformed by other configurations; *ParamILS* uses a powerful stochastic local search (SLS) method to search within potentially vast spaces of can-

didate configurations of a given algorithm; and sequential model-based optimisation (SMBO) methods build a response surface model that relates parameter settings to performance, and use this model to iteratively identify promising settings. We also give a brief overview of other algorithm configuration and parameter tuning procedures and comment on the applications in which various methods have proven to be effective.

While we deliberately limit the scope of our discussion to the algorithm configuration problem defined earlier in this section, we note that there are several conceptually closely related problems that arise in the computer-aided design of algorithms (see also 28): *per-instance algorithm selection methods* choose one of several target algorithms to be applied to a given problem instance based on properties of that instance determined just before attempting to solve it (see, e.g., 57, 46, 24, 67, 68); similarly, *per-instance algorithm configuration methods* use instance properties to determine the specific configuration of a parameterised target algorithm to be used for solving a given instance (see, e.g., 34). *Reactive search procedures*, *on-line algorithm control methods* and *adaptive operator selection techniques* switch between different algorithms, heuristic mechanisms and parameter configurations while running on a given problem instance (see, e.g., 14, 10, 16); and *dynamic algorithm portfolio approaches* repeatedly adjust the allocation of CPU shares between algorithms that are running concurrently on a given problem instance (see, e.g., 20).

Furthermore, we attempt neither to cover all algorithm configuration methods that can be found in the literature, nor to present all details of the procedures we describe; instead, we focus on three fundamental approaches of algorithm configuration methods and survey a number of prominent methods based on these, including the state-of-the-art procedures at the time of this writing. We briefly discuss selected applications of these procedures to illustrate their scope and performance, but we do not attempt to give a complete or detailed account of the empirical results found in the literature.

3.2 Racing Procedures

Given a number of candidate solvers for a given problem, the concept of racing is based on a simple yet compelling idea: sequentially evaluate the candidates on a series of benchmark instances and eliminate solvers as soon as they have fallen too far behind the current leader, i.e., the candidate with the overall best performance at a given stage of the race.

Racing procedures were originally introduced for solving model selection problems in machine learning. The first such technique, dubbed *Hoeffding Races* [48], was introduced in a supervised learning scenario, where a black-box learner is evaluated by measuring its error on a set of test instances. The key idea is to test a given set of models, one test instance at a time, and to discard models as soon as they are shown to perform significantly worse than the best ones. Performance is measured as error over all test instances evaluated so far, and models are eliminated

from the race using non parametric bounds on the true error, determined based on Hoeffding's inequality (which gives an upper bound on the probability of the sum of random variables deviating from its expected value). More precisely, a model is discarded from the race if the lower bound on its true error (for a given confidence level $1 - \delta$) is worse than the upper bound on the error of the currently best model. As a result, the computational effort expended in evaluating models becomes increasingly focussed on promising candidates, and the best candidate models end up getting evaluated most thoroughly.

This idea can be easily transferred to the problem of selecting an algorithm from a set of candidates, where each candidate may correspond to a configuration of a parameterised algorithm [11]. In this context, candidate algorithms (or configurations) are evaluated on a given set of problem instances. As in the case of model selection, the race proceeds in steps, where in each step every candidate is evaluated on the same instance, taken from the given instance set, and candidates that performed significantly worse on the instances considered so far are eliminated from the race. (We note that the evaluation of candidates in each step can, in principle, be performed independently in parallel.)

This procedure requires that the set of candidate algorithms be finite and, since in the initial steps of a race all candidates will need to be evaluated, of somewhat reasonable size. Therefore, when applied to algorithm configuration or parameter tuning scenarios with continuous parameters, racing approaches need to make use of discretisation or sampling techniques. In the simplest case, all continuous parameters are discretised prior to starting the race. Alternatively, stages of sampling and racing can be interleaved, such that the candidate configurations being considered become increasingly concentrated around the best performing configurations.

In the following, we will first present the F-Race procedure of Birattari et al. [11] in more detail and outline its limitations. We will then discuss variations of F-Race that overcome those weaknesses [5, 12], and finally summarise some results achieved by these racing procedures in various algorithm configuration scenarios.

3.2.1 F-Race

The F-Race algorithm by Birattari et al. [11] closely follows the previously discussed racing procedure. Similarly to Hoeffding races, it uses a non parametric test as the basis for deciding which configurations to eliminate in any given step. However, rather than just performing pairwise comparisons with the currently best configuration (the so-called *incumbent*), F-Race first uses the rank-based Friedman test (also known as *Friedman two-way analysis of variance by ranks*) for ni independent s-variate random variables, where s is the number of configurations still in the race, and ni is the number of problem instances evaluated so far. The Friedman test assesses whether the s configurations show no significant performance differences on the ni given instances; if this null hypothesis is rejected, i.e., if there is evidence that some configurations perform better than others, a series of pairwise

```
procedure F-Race
    input target algorithm A, set of configurations C, set of problem instances I,
          performance metric m;
    parameters integer ni_min;
    output set of configurations C*;
    C* := C; ni := 0;
    repeat
        randomly choose instance i from set I;
        run all configurations of A in C* on i;
        ni := ni + 1;
        if ni ≥ ni_min then
            perform rank-based Friedman test on results for configurations in C* on all instances
                in I evaluated so far;
            if test indicates significant performance differences then
                c* := best configuration in C* (according to m over instances evaluated so far);
                for all c ∈ C* \ {c*} do
                    perform pairwise Friedman post hoc test on c and c*;
                    if test indicates significant performance differences then
                        eliminate c from C*;
                    end if;
                end for;
            end if;
        end if;
    until termination condition met;
    return C*;
end F-Race
```

Fig. 3.1: Outline of F-Race for algorithm configuration (original version, according to 11). In typical applications, ni_{min} is set to values between 2 and 5; further details are explained in the text. When used on its own, the procedure would typically be modified to return $c^* \in C^*$ with the best performance (according to m) over all instances evaluated within the race

post hoc tests between the incumbent and all other configurations is performed. All configurations found to have performed significantly worse than the incumbent are eliminated from the race. An outline of the F-Race procedure for algorithm configuration, as introduced by [11], is shown in Figure 3.1; as mentioned by [5], runs on a fixed number of instances are performed before the Friedman test is first applied. The procedure is typically terminated either when only one configuration remains, or when a user-defined time budget has been exhausted.

The Friedman test involves ranking the performance results of each configuration on a given problem instance; in the case of ties, the average of the ranks that would have been assigned without ties is assigned to each tied value. The test then determines whether some configurations tend to be ranked better than others when considering the rankings for all instances considered in the race up to the given iteration. Following Birattari et al. [11], we note that performing the ranking separately for each problem instance amounts to a blocking strategy on instances. The use of

this strategy effectively reduces the impact of noise effects that may arise from the performance variation observed over the given instances set for any configuration of the target algorithm under consideration; this can become critical when those performance variations are large, as has been observed for many algorithms for various hard combinatorial problems (see, e.g., 21, 29).

3.2.2 Sampling F-Race and Iterative F-Race

A major limitation of this basic version of F-Race stems from the fact that in the initial steps all given configurations have to be evaluated. This property of basic F-Race severely limits the size of the configuration spaces to which the procedure can be applied effectively – particularly when dealing with configuration spaces corresponding to so-called *full factorial designs*, which contain all combinations of values for a set of discrete (or discretised) parameters. Two more recent variants of F-Race, Sampling F-Race and Iterative F-Race, have been introduced to address this limitation [5]; both use the previously described F-Race procedure as a subroutine.

Sampling F-Race (short: *RSD/F-Race*) is based on the idea of using a sampling process to determine the initial set of configurations subsequently used in a standard F-Race. In RSD/F-Race, a fixed number r of samples is determined using a so-called *Random Sampling Design*, in which each configuration is drawn uniformly at random from the given configuration space C. (In the simplest case, where no conditional parameters or forbidden configurations exist, this can be done by sampling values for each parameter independently and uniformly at random from the respective domain.) As noted by Balaprakash et al. [5], the performance of this procedure depends substantially on r, the number of configurations sampled in relation to the size of the given configuration space.

A somewhat more effective approach for focussing a procedure based on F-Race on promising configurations is *Iterative F-Race* (short: *I/F-Race*). The key idea behind I/F-Race is the use of an iterative process, where in the first stage of each iteration configurations are sampled from a probabilistic model M, while in the second stage a standard F-Race is performed on the resulting sample, and the configurations surviving this race are used to define or update the model M used in the following iteration. (See Figure 3.2.)

The probabilistic model used in each iteration of I/F-Race consists of a series of probability distributions, $\mathscr{D}_1, \ldots, \mathscr{D}_s$, each of which is associated with one of s 'promising' parameter configurations, $c_1, \ldots, c_s$. Balaprakash et al. [5] consider only numerical parameters and define each distribution $\mathscr{D}_i$ to be a k-variate normal distribution $\mathscr{N}_i := \mathscr{N}(\mu_i, \Sigma_i)$ that is centred on configuration c_i, i.e., $\mu_i = c_i$. They further define the covariance between any two different parameters in a given $\mathscr{N}_i$ to be zero, such that $\mathscr{N}_i$ can be factored into k independent, univariate normal distributions. To start the process with an unbiased probabilistic model, in the first iteration of I/F-Race a single k-variate uniform distribution is used, which is defined as the product of the k independent uniform distributions over the ranges of each given parameter

procedure *I/F-Race*
  input *target algorithm A, set of configurations C, set of problem instances I, performance metric m*;
  output *set of configurations* C^*;
  initialise probabilistic model M;
  $C' := \emptyset$; *// later,* C' *is the set of survivors from the previous F-Race*
  repeat
    based on model M, sample set of configurations $\widehat{C} \subseteq C$;
    perform F-Race on configurations in $\widehat{C} \cup C'$ to obtain set of configurations C^*;
    update probabilistic model M based on configurations in C^*;
    $C' := C^*$;
  until termination condition met;
  return $c^* \in C^*$ with best performance (according to m) over all instances evaluated;
end *I/F-Race*

Fig. 3.2: High-level outline of Iterated F-Race, as introduced by [5]; details are explained in the text. The most recent version of I/F-Race slightly deviates from this outline (see 12)

(we note that this can be seen as a degenerate case of the normal distributions used subsequently, in which the variance is infinite and truncation is applied).

In each iteration of I/F-Race, a certain number of configurations are sampled from the distributions $\mathcal{N}_1, \ldots, \mathcal{N}_s$. In the first iteration, this corresponds to sampling configurations uniformly at random from the given configuration space. In subsequent iterations, for each configuration to be sampled, first, one of the $\mathcal{N}_i$ is chosen using a rank-based probabilistic selection scheme based on the performance of the configuration c_i associated with $\mathcal{N}_i$ (for details, see 5), and then a configuration is sampled from this distribution. Values that are outside the range allowable for a given parameter are set to the closer of the two boundaries, and settings for parameters with integer domains are rounded to the nearest valid value. The number a of configurations sampled in each iteration depends on the number s of configurations that survived the F-Race in the previous iteration; Balaprakash et al. [5] keep the overall number of configurations considered in each iteration of I/F-Race constant at some value r, and therefore simply replace those configurations eliminated by F-Race with newly sampled ones (i.e., $a := r - s$, where in the first iteration, $s = 0$).

The resulting population of $a + s$ configurations is subjected to a standard F-Race; this race is terminated using a complex, disjunctive termination condition that involves a (lower) threshold on the number of surviving configurations as well as upper bounds on the computational budget (measured in target algorithm runs) and the number of problem instances considered [2]. Each of the F-Races conducted within I/F-Race uses a random permutation of the given instance set in order to

[2] The threshold mechanism ends the race as soon as the number of survivors has fallen below k, the number of target algorithm parameters.

avoid bias due to a particular instance ordering. The s configurations that survived the race (where the value of s depends on the part of the termination condition that determined the end of that race) induce the probabilistic model used in the following iteration of I/F-Race.

To increasingly focus the sampling process towards the most promising configurations, the standard deviations of the component distributions of the probabilistic models $\mathcal{N}_i$ are gradually decreased using a volume reduction technique. More precisely, after each iteration, the standard deviation vector σ_i of each distribution $\mathcal{N}_i$ is scaled by a factor $(1/r)^k$, where r is the total number of configurations entered into the F-Race, and k is the number of given parameters; this corresponds to a reduction of the total volume of the region bounded by $\mu_i \pm \sigma_i$ (over all k parameters) by a factor of r. At the beginning of I/F-Race, when configurations are sampled uniformly, the standard deviation values are (somewhat arbitrarily) set to half of the range of the respective parameter values.

I/F-Race, as specified by Balaprakash et al. [5], assumes that all parameters are numerical. This limitation is overcome in a later variant [12], which supports categorical parameters by sampling their values from discrete probability distributions that are updated by redistributing probability mass to values seen in good configurations, as determined by F-Race. This version of I/F-Race, which we call *I/F-Race-10* for clarity, also differs from the one described previously in several other aspects. Notably, the number of iterations in I/F-Race-10 is determined as $2 + \lfloor \log_2(k) + 0.5 \rfloor$, and the overall computational budget (i.e., number of target algorithm runs) is distributed equally over these iterations. Furthermore, the number r of configurations considered at iteration number t is set to $\lfloor b/(5+t) \rfloor$, where b is the computational budget available for that iteration; this leads to fewer configurations being considered in later iterations. The threshold on the number of survivors below which any given F-Race is terminated is also determined as $2 + \lfloor \log_2(k) + 0.5 \rfloor$. Finally, I/F-Race-10 handles conditional parameters by only sampling values for them when they are active, and by only updating the respective component of the model in situations where such parameters are active in a configuration surviving one of the subsidiary F-Races. (For further details, see 12.)

3.2.3 Applications

Balaprakash et al. [5] describe applications of F-Race, Sampling F-Race and Iterative F-Race to three high-performance stochastic local search algorithms: MAX-MIN Ant System for the TSP with six parameters [64], an estimation-based local search algorithm for the probabilistic TSP (PTSP) with three parameters [6], and a simulated annealing algorithm for vehicle routing with stochastic demands (VRP-SD) with four parameters [53]. The empirical results from these case studies indicate that both, Sampling F-Race and Iterative F-Race can find good configurations in spaces that are too big be handled effectively by F-Race, and that Iterative F-Race tends to give better results than Sampling F-Race, especially when applied to

more difficult configuration problems. Both, the PTSP and the VRP-SD algorithms as configured by Iterative F-Race represented the state of the art in solving these problems at the time of this study.

More applications of F-Race have recently been summarised by Birattari et al. [12]. These include tuning the parameters of various meta-heuristic algorithms for university timetabling problems [58], of a control system for simple robots [52], and of a new state-of-the-art memetic algorithm for the linear ordering problem [61]. In all of these cases, the basic F-Race algorithm was applied to target algorithms with few parameters and rather small configuration spaces (48–144 configurations).

Yuan et al. [71] report an application of I/F-Race for tuning various heuristic algorithms for solving a locomotive scheduling problem provided by the German railway company, Deutsche Bahn. The target algorithms considered in this work had up to five parameters, mostly with continuous domains. The most complex application of I/F-Race reported by Birattari et al. [12] involves 12 parameters of the ACOTSP software, some of which conditionally depend on the values of others.

While these (and other) racing procedures have been demonstrated to be useful for accomplishing a broad range of parameter tuning tasks, it is somewhat unclear how well they perform when applied to target algorithms with many more parameters, and how effectively they can deal with the many categorical and conditional parameters arising in the context of more complex computer-aided algorithm design tasks, such as the ones considered by Hutter et al. [35], KhudaBukhsh et al. [45], Hutter et al. [42], and Tompkins and Hoos [66].

3.3 ParamILS

When manually solving algorithm configuration problems, practitioners typically start from some configuration (often default or arbitrarily chosen settings) and then attempt to achieve improvements by modifying one parameter value at a time. If such an attempt does not result in improved performance, the modification is rejected and the process continues from the previous configuration. This corresponds to an iterative first-improvement search in the space of configurations.

While the idea of performing local search in configuration space is appealing considering the success achieved by similar methods on other hard combinatorial problems, iterative improvement is a very simplistic method that is limited to finding local optima. The key idea behind ParamILS is to combine more powerful stochastic local search (SLS) methods with mechanisms aimed at exploiting specific properties of algorithm configuration problems. The way in which this is done does not rely on an attempt to construct or utilise a model of good parameter configurations or of the impact of configuations on target algorithm perfomance; therefore, ParamILS is a *model-free search procedure*.

3.3.1 The ParamILS Framework

At the core of the ParamILS framework for automated algorithm configuration [36, 37] lies *Iterated Local Search (ILS)*, a well-known and versatile stochastic local search method that has been applied with great success to a wide range of difficult combinatorial problems (see, e.g., 47, 30). ILS iteratively performs phases of simple local search designed to rapidly reach or approach a locally optimal solution to the given problem instance, interspersed with so-called perturbation phases, whose purpose is to effectively escape from local optima. Starting from a local optimum x, in each iteration one perturbation phase is performed, followed by a local search phase, with the aim of reaching (or approaching) a new local optimum x'. Then, a so-called acceptance criterion is used to decide whether to continue the search process from x' or whether to revert to the previous local optimum, x. Using this mechanism, ILS aims to solve a given problem instance by effectively exploring the space of its locally optimal solutions. At a lower level, ILS – like most SLS methods – visits (i.e., moves through) a series of candidate solutions such that at any given time there is a current candidate solution, while keeping track of the incumbent (i.e., the best solution encountered so far).

ParamILS uses this generic SLS method to search for high-performance configurations of a given algorithm as follows (see also Figure 3.3). The search process is initialised by considering a given configuration (which would typically be the given target algorithm's default configuration) as well as r further configurations that are chosen uniformly at random from the given configuration space. These $r+1$ configurations are evaluated in a way that is specific to the given ParamILS variant, and the best-performing configuration is selected as the starting point for the iterated local search process. This initialisation mechanism can be seen as a combination of the intuitive choice of starting from a user-defined configuration (such as the target algorithm's default settings) and a simple experimental design technique, where the latter makes it possible to exploit situations where the former represents a poor choice for the given set of benchmark instances. Clearly, there is a trade-off between the effort spent on evaluating randomly sampled configurations at this point and the effort used in the subsequent iterated local search process. Hutter et al. [39] reported empirical results suggesting that $r = 10$ results in better performance than $r = 0$ and $r = 100$ across a number of configuration scenarios. However, we suspect that more sophisticated initialisation procedures, in particular ones based on racing or sequential model-based optimisation techniques, might result in even better performance.

The subsidiary local search procedure used in ParamILS is based on the one-exchange neighbourhood induced by arbitrary changes in the values of a single target algorithm parameter. ParamILS supports conditional parameters by pruning neighbourhoods such that changes in inactive parameters are excluded from consideration; it also supports exclusion of (complete or partial) configurations explicitly declared 'forbidden' by the user. Using the one-exchange neighbourhood, ParamILS performs iterative first-improvement search – an obvious choice, considering the computational cost of evaluating candidate configurations. We believe that larger neighbourhoods might prove effective in situations in which parame-

```
procedure ParamILS
    input target algorithm A, set of configurations C, set of problem instances I,
          performance metric m;
    parameters configuration c_0 ∈ C, integer r, integer s, probability pr;
    output configuration c*;

    c* := c_0;
    for i := 1 to r do
        draw c from C uniformly at random;
        assess c against c* based on performance of A on instances from I according to metric m;
        if c found to perform better than c* then
            c* := c;
        end if;
    end for;

    c := c* ;
    perform subsidiary local search on c;
    while termination condition not met do
        c' := c;
        perform s random perturbation steps on c'
        perform subsidiary local search on c';
        assess c' against c based on performance of A on instances from I according to metric m;
        if c' found to perform better than c then        // acceptance criterion
            update overall incumbent c*;
            c := c';
        end if;
        with probability pr do
            draw c from C uniformly at random;
        end with probability;
    end while;
    return c*;
end ParamILS
```

Fig. 3.3: High-level outline of ParamILS, as introduced by [36]; details are explained in the text

ter effects are correlated, as well as in conjunction with mechanisms that recognise and exploit such dependencies in parameter response. Furthermore, search strategies other than iterative first-improvement could be considered in variants of ParamILS that build and maintain reasonably accurate models of local parameter responses.

The perturbation procedure used in the ParamILS framework performs a fixed number, s, of steps chosen uniformly at random in the same one-exchange neighbourhood used during the local search phases. Computational experiments in which various fixed values of s as well as several multiples of the number of target algorithm parameters were considered suggest that relatively small perturbations (i.e., $s = 2$) are sufficient for obtaining good performance of the overall configuration procedure [39]. Considering the use of iterative first-improvement during the local search phases, this is not overly surprising; still, larger perturbations might be effec-

tive in combination with model-based techniques within the ParamILS framework in the future.

While various acceptance criteria have been used in the literature on ILS, ParamILS uses one of the simplest mechanisms: Between two given candidate configurations, it always chooses the one with better observed performance; ties are broken in favour of the configuration reached in the most recent local search phase. This results in an overall behaviour of the iterated local search process equivalent to that of an iterative first-improvement procedure searching the space of configurations reached by the subsidiary local search process. Considering once again the computational effort involved in each iteration of ParamILS, this is a natural choice; however, in cases where many iterations of ParamILS can be performed, and where the given configuration space contains attractive regions with many local minima, more complex acceptance criteria that provide additional search diversification (e.g., based on the Metropolis criterion) might prove useful.

In addition to the previously described perturbation procedure, ParamILS also uses another diversification mechanism: At the end of each iteration, with a fixed probability pr (by default set to 0.01), the current configuration is abandoned in favour of a new one that is chosen uniformly at random and serves as the starting point for the next iteration of the overall search process. This restart mechanism provides the basis for the probabilistic approximate completeness of FocusedILS, the more widely used of the two ParamILS variants discussed in the following. We believe that it also plays an important role towards achieving good performance in practice, although anecdotal empirical evidence suggests that additional diversification of the search process is required in order to eliminate occasionally occurring stagnation behaviour.

Finally, like the racing procedures discussed in the previous section, ParamILS performs blocking on problem instances, i.e., it ensures that comparisons between different configurations are always based on the same set of instances. This is important, since the intrinsic hardness of problem instances for any configuration of the given target algorithm may differ substantially. Furthermore, when used for optimising the performance of a randomised target algorithm A, ParamILS also blocks on the pseudo random number seeds used in each run of A; the main reason for this lies in our desire to avoid spurious performance differences in cases where the differences between two configurations have no impact on the behaviour of A.

3.3.2 BasicILS

The conceptually simplest way of assessing the performance of a configuration of a given target algorithm A is to perform a fixed number of runs of A. This is precisely what happens in BasicILS(N), where the user-defined parameter N specifies the number of target algorithm runs performed for each configuration to be assessed, using the same instances and pseudo random number seeds. Applied to a randomised target algorithm A, BasicILS(N) will only perform multiple runs per instance if N

exceeds the number of given problem instances; in this case, the list of runs performed is determined by a sequence of random permutations of the given set of instances, and the random number seed used in each run is determined uniformly at random.

This approach works well for configuration scenarios where a relatively small set of benchmark instances is representative of all instances of interest. Furthermore, the N target algorithm runs per configuration can be performed independently in parallel. As for all ParamILS variants – and, indeed, for any SLS algorithm – further parallelisation can be achieved by performing multiple runs of BasicILS(N) in parallel. Finally, in principle, it is possible to perform multiple parallel runs of the subsidiary local search in each iteration or to evaluate multiple neighbours of a configuration in each search step independently in parallel.

3.3.3 FocusedILS

One drawback of BasicILS is that it tends to make substantial effort evaluating poor configurations, especially when used to configure a given target algorithm for minimised run-time. The only way to reduce that effort is to choose a small number of runs, N; however, this can (and often does) result in poor generalisation of performance to problem instances other than those used during the configuration process. FocusedILS addresses this problem by initially evaluating configurations using few target algorithm runs and subsequently performing additional runs to obtain increasingly precise performance estimates for promising configurations. We note that the idea of focussing the computational effort in evaluating configurations on candidates that have already shown promising performance is exactly the same as that underlying the concept of racing. However, unlike the previously discussed racing procedures, FocusedILS determines promising configurations heuristically rather than using statistical tests.

The mechanism used by FocusedILS to assess configurations is based on the following concept of *domination*: Let c_1 and c_2 be configurations for which $N(c_1)$ and $N(c_2)$ target algorithm runs have been performed, respectively. As in the case of BasicILS, the runs performed for each configuration follow the same sequence of instances (and pseudo random number seeds). Then c_1 dominates c_2 if, and only if, $N(c_1) \geq N(c_2)$ and the performance estimate for c_1 based on its first $N(c_2)$ runs is at least as good as that for c_2 based on all of its $N(c_2)$ runs. This definition incorporates the previously discussed idea of blocking, as configurations are compared based on their performance on a common set of instances (and pseudo random number seeds).

Whenever FocusedILS decides that one configuration, c_1, performs better than another, c_2, it ensures that c_1 dominates c_2 by performing additional runs on either or both configurations. More precisely, when comparing two configurations, an additional run is first performed for the configuration whose performance estimate is based on fewer runs or, in the case of a tie, on both configurations. Then, as long as neither configuration dominates the other, further runs are performed based on the

same criterion. Furthermore, when domination has been determined, FocusedILS performs additional runs for the winner of the comparison (ties are always broken in favour of more recently visited configurations). The number of these *bonus runs* is determined as the number of configurations visited since the last improving search step, i.e., since the last time a comparison between two configurations was decided in favour of the one that had been visited more recently. This mechanism ensures that the better a configuration appears to perform, the more thoroughly it is evaluated, especially in cases where a performance improvement is observed after a number of unsuccessful attempts.

As first established by Hutter et al. [36], FocusedILS has an appealing theoretical property: With increasing run-time, the probability of finding a configuration with globally optimal performance on the given set of benchmark instances approaches 1. This probabilistic approximate completeness (PAC) property follows from two key observations: Firstly, thanks to the previously mentioned probabilistic restart mechanism used in the ParamILS framework, over time any configuration from a finite configuration space is visited arbitrarily often. Secondly, as the number of visits to a given configuration increases, so does the number of target algorithm runs FocusedILS performs on it, which causes the probability of mistakenly failing to recognise that its true performance on the given instance set is better than that of any other configuration it is compared with approach 0. While this theoretical guarantee only concerns the behaviour of FocusedILS at the limit, as run-time approaches infinity, the mechanisms giving rise to it appear to be very effective in practice when dealing with surprisingly large configuration spaces (see, e.g., 35, 37, 42). Nevertheless, stagnation of the search process has been observed in several cases and is typically ameliorated by performing multiple runs of FocusedILS independently in parallel, from which a single winning configuration is determined based on the performance observed on the set of benchmark instances used in those runs. We expect that by replacing the simplistic probabilistic restart mechanism, and possibly modifying the mechanism used for allocating the additional target algorithm runs to be performed when assessing configurations, stagnation can be avoided or overcome more effectively.

3.3.4 Adaptive Capping

Both BasicILS and FocusedILS can be improved by limiting under certain conditions the time that is spent evaluating poorly performing configurations. The key idea is that when comparing two configurations c_1 and c_2, a situation may arise where, regardless of the results of any further runs, c_2 cannot match or exceed the performance of c_1 [37]. This is illustrated by the following example, taken from Hutter et al. [37]: Consider the use of BasicILS(100) for minimising the expected run-time of a given target algorithm on a set of 100 benchmark instances, where configuration c_1 has solved all 100 instances in a total of ten CPU seconds, and c_2 has run for the same ten CPU seconds on the first instances without solving them.

Clearly, we can safely terminate that latter run after $10 + \varepsilon$ CPU seconds (for some small time ε), since the average run-time of c_2 must exceed 0.1 CPU seconds, regardless of its performance in the remaining $N - 1$ runs, and therefore be worse than that of c_1.

Based on this insight, the *trajectory-preserving adaptive capping mechanism* of Hutter et al. [37] limits the effort spent on evaluating configurations based on comparing lower bounds on the performance of one configuration c_2 to upper bounds (or exact values) on that of another configuration c_1, based on the results of given sets of runs for c_1 and c_2. We note that this corresponds to the notion of racing, where each of the two configurations works independently through a given number of runs, but the race is terminated as soon as the winner can be determined with certainty. Apart from the potential for savings in running time, the use of trajectory-preserving capping does not change the behaviour of ParamILS.

A heuristic generalisation of this capping mechanism makes it possible to achieve even greater speedups, albeit at the price of possibly substantial changes to the search trajectory followed by the configuration procedure. The key idea behind this generalisation (dubbed *aggressive capping*) is to additionally bound the time allowed for evaluating configurations based on the performance observed for the current incumbent, i.e., the best-performing configuration encountered since the beginning of the ParamILS run. The additional bound is obtained by multiplying the performance estimate of the incumbent by a constant, bm, called the *bound multiplier*. Formally, for $bm = \infty$, the additional bound becomes inactive (assuming the performance measure is to be minimised), and the behaviour of trajectory-preserving capping is obtained. For $bm = 1$, on the other hand, a very aggressive heuristic is obtained, which limits the evaluation of any configuration to the time spent on evaluating the current incumbent. In practice, $bm = 2$ appears to result in good performance and is used as a default setting in ParamILS. Despite its heuristic nature, this modified capping mechanism preserves the PAC property of FocusedILS.

Although Hutter et al. [37] spelled out their adaptive capping mechanisms for the performance objective of minimising a target algorithm's mean run-time only, these mechanisms generalise to other objectives in a rather straightforward way (a discussion of capping in the context of minimising quantiles of run-time is found in Ch. 7 of [32]). We note, however, that – especially when several target algorithm runs are conducted in parallel – adaptive capping would be most effective in the case of run-time minimisation. Particularly substantial savings can be achieved during the assessment of the $r + 1$ configurations considered during initialisation of the search process, as well as towards the end of each local search phase. Finally, it should be noted that adaptive capping mechanisms can be used in the context of configuration procedures other than ParamILS; for example, Hutter et al. [37] mention substantial speedups achieved by using adaptive capping in combination with simple random sampling (the same procedure as that used during the initialisation of ParamILS).

3.3.5 Applications

ParamILS variants, and in particular FocusedILS, have been very successfully applied to a broad range of high-performance algorithms for several hard combinatorial problems. An early version of FocusedILS was used by Thachuk et al. [65] to configure a replica-exchange Monte Carlo (REMC) search procedure for the 2D and 3D HP protein structure prediction problems; the performance objective was to minimise the mean run-time for finding ground states for a given set of sequences in these abstract but prominent models of protein structure, and the resulting configurations of the REMC procedure represented a considerable improvement over the state of the art techbniques for solving these challenging problems.

FocusedILS has also been used in a series of studies leading to considerable advances in the state-of-the-art in solving the satisfiability problem in propositional logic, one of the most widely studied $\mathcal{NP}$-hard problems in computer science. Hutter et al. [35] applied this procedure to SPEAR, a complete, DPLL-type SAT solver with 26 parameters (ten of which are categorical), which jointly give rise to a total of $8.34 \cdot 10^{17}$ possible configurations. The design of SPEAR was influenced considerably by the availability of a powerful configuration tool such as FocusedILS, whose application ultimately produced configurations that solved a given set of SAT-encoded software verification problems about 100 times faster than previous state-of-the-art solvers for these types of SAT instances and won the first prize in the QF_BV category of the 2007 Satisfiability Modulo Theories (SMT) Competition.

KhudaBukhsh et al. [45] used FocusedILS to find performance-optimised instantiations of SATenstein-LS, a highly parametric framework for stochastic local search (SLS) algorithms for SAT. This framework was derived from components found in a broad range of high-performance SLS-based SAT solvers; its 41 parameters induce a configuration space of size $4.82 \cdot 10^{12}$. Using FocusedILS, performance improvements of up to three orders of magnitudes were achieved over the previous best-performing SLS algorithms for various types of SAT instances, for several of which SLS-based solvers are the most effective SAT algorithms overall. Several automatically determined configurations of SATenstein-LS were used in the most recent SATzilla solvers, which led the field in the 2009 SAT Competition, winning prizes in five of the nine main categories [69].

Very recently, Xu et al. [70] used FocusedILS in an iterative fashion to obtain sets of configurations of SATenstein-LS that were then used in combination with state-of-the-art per-instance algorithm selection techniques (here: SATzilla). In each iteration of the overall procedure, dubbed *Hydra*, FocusedILS was used to find configurations that would best complement a given portfolio-based per-instance algorithm selector. This approach resulted in a portfolio-based SAT solver that, while derived in a fully automated fashion from a single, highly parameterised algorithm, achieved state-of-the-art performance across a wide range of benchmark instances.

Tompkins and Hoos [66] applied FocusedILS to a new, flexible framework for SLS-based SAT solvers called VE-Sampler (which is conceptually orthogonal to the previously mentioned SATenstein-LS framework). VE-Sampler has a large number of categorical and conditional parameters, which jointly give rise to more than 10^{50}

distinct configurations, and using FocusedILS, configurations could be found that were shown to solve two well-known sets of SAT-encoded software verification problems between 3.6 and nine times faster than previous state-of-the-art SLS-based SAT solvers for these types of SAT instances.

Chiarandini et al. [15] used FocusedILS to configure a hybrid, modular SLS algorithm for a challenging university timetabling problem that subsequently placed third in Track 2 of the Second International Timetabling Competition (ITC 2007). The configuration space considered in this context was relatively small (seven parameters, 50,400 configurations), but the use of automated algorithm configuration made it possible to achieve close to state-of-the-art performance in very limited human development time and without relying on deep and extensive domain expertise. In subsequent work, Fawcett et al. [17] expanded the design space by parameterising additional parts of the solver, and – using multiple rounds of FocusedILS with different performance objectives – obtained a configuration that was demonstrated to substantially improve upon the state of the art for solving the post-enrolment course timetabling problem considered in Track 2 of ITC 2007. The overall performance metric used in these studies (and in the competition) was solution quality achieved by the target solver after a fixed amount of time.

Recently, Hutter et al. [42] reported substantial improvements in the performance of several prominent solvers for mixed integer programming (MIP) problems, including the widely used industrial CPLEX solver. In the case of CPLEX, FocusedILS was used to configure 76 parameters, most of which are categorical (and some conditional), giving rise to a configuration space of size $1.9 \cdot 10^{47}$. Despite the fact that the default parameter settings for CPLEX are known to have been chosen carefully, based on a considerable amount of thorough experimentation, substantial performance improvements were obtained for many prominent types of MIP instances, in terms of both time required for finding optimal solutions (and proving optimality) and the minimising of the solution quality (optimality gap) achieved within a fixed amount of time (speedup factors beween 1.98 and 52.3, and gap reduction factors between 1.26 and 8.65). Similarly impressive results were achieved for another commercial MIP solver, Gurobi, and a prominent open-source MIP solver, lpsolve.

Finally, Hutter et al. [39] reported a successful meta-configuration experiment, in which BasicILS was used to optimise the four parameters that control the behaviour of FocusedILS. BasicILS was chosen as the meta-configurator, since its runs can be parallelised in a straightforward manner. At least partly because of the enormous computational cost associated with this experiment (where each target algorithm run corresponds to solving an algorithm configuration task, and hence to executing many costly runs of the algorithm configured in that task, which itself solved instances of an $\mathcal{NP}$-hard problem such as SAT), only marginal improvements in the performance of the configurator, FocusedILS, on a number of previously studied configuration benchmarks could be achieved.

The target algorithms considered in most of these applications have continuous parameters, and up to this point ParamILS requires these parameters to be discretised. While in principle finding reasonable discretisations (i.e., ones whose use does

not cause major losses in the performance of the configurations found by ParamILS) could be difficult, in most cases generic approaches such as even or geometric subdivisions of a given interval seem to give good results. Where this is not the case, multiple runs of the configuration procedure can be used to iteratively refine the domains of continuous parameters. The same approach can be used to extend domains in cases where parameter values in an optimised configuration lie at the boundary of their respective domains. Nevertheless, the development of ParamILS variants that natively deal with continuous parameters and support dynamic extensions of parameter domains remains an interesting direction for future work.

3.4 Sequential Model-Based Optimisation

A potential disadvantage of the model-free search approach underlying ParamILS is that it makes no explicit attempt to recognise and benefit from regularities in a given configuration space. The *model-based search paradigm* underlying the approach discussed in this section, on the other hand, uses the information gained from the configurations evaluated so far to build (and maintain) a model of the configuration space, based on which configurations are chosen to be evaluated in the future. We note that, as also pointed out by its authors, the Iterative F-Race procedure (I/F-Race) discussed in Section 3.2.2 of this chapter is a model-based configuration procedure in this sense. But unlike in I/F-Race, the models used by the methods discussed in the following capture directly the dependency of target algorithm performance on parameter settings. These *response surface models* can be used as surrogates for the actual parameter response of a given target algorithm and provide the basis for determining promising configurations at any stage of an iterative model-based search procedure; this generic approach is known as *sequential model-based optimisation (SMBO)* and can be seen as a special case of sequential analysis – a broader area within statistics that also comprises sequential hypothesis testing and so-called multi-armed bandits. An outline of SMBO for algorithm configuration is shown in Figure 3.4; in principle, performance measurements for multiple configurations can be performed independently in parallel.

The setting considered in almost all work on sequential model-based optimisation procedures up to this day is known as the *black-box optimisation problem*: Given an unknown function f and a space of possible inputs X, find an input $x \in X$ that optimises f based on measurements of f on a series of inputs. The function to be optimised, f, may be deterministic or stochastic; in the latter case, measurements are subject to random noise, and formally the values of f are random variables. Algorithm configuration can be seen as a special case of black-box optimisation, where the function to be optimised is the performance m of an algorithm A on a set of problem instances I. However, in contrast to algorithm configuration procedures such as FocusedILS or F-Race, black-box optimisation procedures do not take into account the fact that approximate measurements can be obtained at lower computational cost by running A on subsets of I, and that, by blocking on instances (and pseudo-random

```
procedure SMBO
    input target algorithm A, set of configurations C, set of problem instances I,
          performance metric m;
    output configuration c*;

    determine initial set of configurations C_0 ⊂ C;
    for all c ∈ C_0, measure performance of A on I according to metric m;
    build initial model M based on performance measurements for C_0;
    determine incumbent c* ∈ C_0 for which best performance was observed or predicted;
    repeat
        based on model M, determine set of configurations C' ⊆ C;
        for all c ∈ C', measure performance of A on I according to metric m;
        update model M based on performance measurements for C';
        update incumbent c*;
    until termination condition met;
    return c*;
end SMBO
```

Fig. 3.4: High-level outline of the general sequential model-based optimisation approach to automated algorithm configuration; model M is used to predict the performance of configurations that have not (yet) been evaluated, and set C' is typically chosen to contain configurations expected to perform well based on those predictions. Details of various algorithms following this approach are explained in the text

number seeds), performance measurements for different configurations can be compared more meaningfully; furthermore, they have no means of exploiting knowledge about the instances in I acquired from earlier target algorithm runs.

Because black-box function optimisation is somewhat more general than algorithm configuration, and methods for solving black-box functions are easily applicable to modelling and optimising the response of a wide range of systems, in the following we use standard terminology from the statistics literature on experimental design, in particular, *design point* for elements of the given input space X and *response* for values of the unknown function f. In the context of algorithm configuration, design points correspond to configurations of a given target algorithm A, and response values represent A's performance m on instance set I. A unified, more technical presentation of the methods covered in this section can be found in the dissertation of Hutter [32], and further details are provided in the original articles referenced throughout.

3.4.1 The EGO Algorithm

The efficient global optimisation (EGO) algorithm for black-box function optimisation by Jones et al. [44] uses a response surface model obtained via noise-free Gaussian process regression in combination with an expected improvement crite-

rion for selecting the next configuration to be evaluated. The noise-free Gaussian process (GP) model utilised by EGO is also known as the *DACE model*, after its prominent use in earlier work by Sacks et al. [59]. It defines for every input x a random variable $\hat{F}(x)$ that characterises the uncertainty over the true response value $f(x)$ at point x.

The model-based optimisation process carried out by EGO starts with about $10 \cdot k$ design points determined using a k-dimensional space-filling Latin hypercube design (LHD). After measuring the response values for these values, the $2 \cdot k + 2$ parameters of a DACE model are fit to the pairs of design points and response values, using maximum likelihood estimates (as described by 44, this can be partially done in closed form). The resulting model is assessed by means of so-called *standardized cross-validated residuals*, which reflect the degree to which predictions made by the model agree with the observed response values on the design points used for constructing the model. If the model is deemed unsatisfactory, the response values may be transformed using a log- or inverse-transform (i.e., modified by applying the function $\ln y$ or $1/y$) and the model fitted again.

After a satisfactory initial model has been obtained, it is used in conjunction with an *expected improvement criterion* to determine a new design point to be evaluated. The expected improvement measure used in this context uses the current DACE model M to estimate the expected improvement over the best response value measured so far, f_{min}, at any given design point x, and is formally defined as $EI(x) := E[\max\{f_{min} - \hat{F}(x), 0\}]$, where $\hat{F}(x)$ is the random variable describing the response for a design point x according to model M. Using a closed-form expression for this measure given by Jones et al. [44] and a branch & bound search method (which can be enhanced heuristically), the EGO algorithm then determines a design point x' with maximal expected improvement $EI(x')$. If $EI(x')$ is less than 1% of the current incumbent, the procedure terminates. Otherwise, the response value $f(x')$ is measured, and the DACE model is refitted on the previous set of data extended by the pair $(x', f(x'))$, and a new iteration begins, in which the updated model is used to determine the next design point using the same process that yielded x'.

Note that in every iteration of this process, the DACE model has to be fitted, which involves a matrix inversion of cost $O(n^3)$, where n is the number of design points used. Depending on the cost of measuring the response value for a given design point, this may represent a substantial computational overhead. Furthermore, the noise-free Gaussian process model used in EGO cannot directly characterise the stochastic responses obtained when solving algorithm configuration problems involving randomised target algorithms.

3.4.2 Sequential Kriging Optimisation and Sequential Parameter Optimisation

We now discuss two black-box optimisation procedures that deal with stochastic responses, as encountered when modelling phenomena subject to observation noise or configuring randomised algorithms.

The first of these, known as *Sequential Kriging Optimisation* [31] estimates a Gaussian process (GP) model (also known as a *kriging model*) directly from samples (i.e., noisy measurements) of response values. Similarly to EGO, SKO starts with an LHD of $10 \cdot k$ design points, for which response values are sampled. To facilitate initial estimates of the observation noise, one additional sample is then drawn for each of the k best of these design points, after which a Gaussian process model M is fitted directly to the resulting set of $11 \cdot k$ pairs of design points and corresponding response values. The resulting model M is then assessed and possibly modified based on a transformation of the response, as in the EGO algorithm.

Next, an incumbent design point is determined based on model M by minimising the expression $\mu(x) + \sigma(x)$ using the Nelder-Mead Simplex algorithm [49], where $\mu(x)$ and $\sigma(x)$ are the mean and standard deviation predicted by M for input x, and the minimisation is over the design points used in constructing the model. This risk adverse strategy is less easily misled by inaccurate estimates of the mean response value than a minimisation of the predicted mean only. The next design point to be evaluated is determined based on model M using an augmented expected improvement measure, designed to steer the process away from design points with low predictive variance. This augmented expected improvement measure is formally defined as

$$EI'(x) := E[\max\{\hat{f}_{min} - \hat{F}(x), 0\}] \cdot \left(1 - \sigma_\varepsilon / \sqrt{s^2(x) - \sigma_\varepsilon^2}\right),$$

where $\hat{f}_{min}$ is the model's prediction for the current best input (as in EGO, obtained by considering all design points used for building the model), $\hat{F}(x)$ is the random variable describing the response for a design point x according to model M, σ_ε is the standard deviation of the measurement noise (assumed to be identical for all inputs), and $s^2(x)$ is the variance of the response $\hat{F}(x)$ given by the model at point x, where the second term in the product decreases as the predictions of M become more accurate. Based on the given model M, the next design point to be evaluated, x', is determined by maximising $EI'(x)$ using the Nelder-Mead Simplex algorithm [49]. Next, the model is refitted, taking into account x' and a response value sampled at x', and a new iteration begins, in which the updated model is used to determine the next design point using the same process that yielded x'. If the maximum $EI'(x)$ values from $d+1$ successive iterations all fall below a user-defined threshold, the iterative sampling process is terminated. (This treshold can be specified as an absolute value or as a fraction of the difference between the largest and smallest observed response values.)

Unfortunately, SKO assumes that the variability of the response values at each design point is characterised by a Gaussian distribution, and that the standard deviations of those distributions are the same across the entire design space. Both of these assumptions are problematic in the context of configuring randomised algorithms, particularly when minimising run-time (see, e.g., 30). Furthermore, the time required for fitting the model in each iteration of SKO is cubic in the number of response values sampled, which can represent a substantial computational burden.

The *Sequential Parameter Optimisation (SPO)* procedure by Bartz-Beielstein et al. [8] follows a fundamentally different strategy to deal with noisy response measurements:[3] Rather than fitting a Gaussian process model directly to a set of sampled responses, for each design point x the measure to be optimised is estimated empirically based on all samples taken at x, and a noise-free Gaussian process model (like the one used in the EGO algorithm) is fitted to the resulting data. In contrast to the approach taken by SKO, this makes it possible to optimise arbitrary statistics of the noisy function values, i.e., in the case of algorithm configuration, of the given target algorithm's run-time distributions; examples of such statistics are the mean, median, and arbitrary quantiles, and also measures of variability, as well as combinations of measures of location and variability. Another advantage of this approach is its substantially lower computational complexity: While SKO requires time cubic in the number of function values sampled, SPO's run-time is only cubic in the number of distinct design points – typically a much lower number.

Like SKO and EGO, SPO uses a Latin hypercube design as a basis for constructing the initial model; however, SPO chooses d design points and samples r response values for each of these, where d and r are specified by the user (the default value of r is 2). Based on these samples, empirical estimates of the measure to be optimised are calculated for each design point, and the point with the best resulting value is chosen as the initial incumbent. Next, a noise-free Gaussian process model is fitted to the resulting set of d pairs of design points and empirical response statistics. This model, M, is sampled for $10,000$ design points chosen uniformly at random,[4] and the best j of these according to an expected improvement (EI) measure are selected for further evaluation, where j is a user-defined number with a default setting of 1. The EI measure used in this context is formally defined as

$$EI^2(x) := E[(f_{min} - \hat{F}(x))^2] = E^2[f_{min} - \hat{F}(x)] + \mathrm{Var}[f_{min} - \hat{F}(x)],$$

where f_{min} is the best value of the measure to be optimised observed so far, and $\hat{F}(x)$ is the distribution over the predictions obtained from model M at design point x. This EI measure has been introduced by Schonlau et al. [62] with the aim of encouraging the exploration of design points for which the current model produces highly uncertain predictions.

At each of the design points determined in this way, r new response values are measured. Furthermore, additional response values are measured for the current incumbent to ensure that it is evaluated based on as many samples as available for any of the new design points. Then, the best of all the design points considered so far, according to the given measure to be optimised, is selected as the new incumbent

[3] In the literature, the term *sequential parameter optimisation* is also used to refer to a broad methodological framework encompassing fully automated as well as interactive approaches for understanding and optimising an algorithm's performance in response to its parameter settings. Here, as in the work of Hutter et al. [38], we use the term more narrowly to refer to the fully automated SMBO procedures implemented in various versions of the Sequential Parameter Optimization Toolbox (SPOT) by [9].

[4] We note that the use of a space-filling design, such as an LHD, should in principle yield better results if implemented sufficiently efficiently.

(with ties broken uniformly at random). If the design point thus selected has been an incumbent at any point earlier in the search process, r is increased; in SPO version 0.3 [8], r is doubled, while in the newer version 0.4 [7], it is merely incremented by 1, and in both cases values of r are limited to a user-specified maximum value r_{max}. At this point, a new iteration of SPO begins, in which a noise-free GP is fitted on the augmented set of data.

3.4.3 *Recent Variants of Sequential Parameter Optimisation: SPO$^+$ and TB-SPO*

Based on a detailed investigation of the core components of the SPO algorithm, Hutter et al. [38] introduced a variant called SPO$^+$ that shows considerably more robust performance on standard benchmarks than the SPO 0.3 and SPO 0.4 algorithms described previously.

The main difference between SPO$^+$ and the previous SPO procedures lies in the way in which new design points are accepted as incumbents. Inspired by FocusedILS, SPO$^+$ uses a mechanism that never chooses a new incumbent $\hat{x}'$ without ensuring that at least as many responses have been sampled at $\hat{x}'$ as at any other design point $x \neq \hat{x}'$. To achieve this, for any challenger to the current incumbent $\hat{x}$, i.e., for any design point x' that appears to represent an improvement over $\hat{x}$ based on the samples taken so far, additional response values are sampled until either x' ceases to represent an improvement, or the number of response values sampled at x' reaches that taken at $\hat{x}$, with x' still winning the comparison with $\hat{x}$ based on the respective samples; only in the latter case does x' become the new incumbent, while in the former case it is dismissed, and as many additional response values are sampled for $\hat{x}$ as newly measured for x'.

The new response values determined for a challenger x' are sampled in batches, with the number of new samples taken doubling in each successive batch. As noted by Hutter et al. [38], using this mechanism, rejection of challengers is done in a rather aggressive, heuristic manner, and frequently occurs after only a single response value has been sampled at x' – long before a statistical test could conclude that the x' is worse than the current incumbent.

The *Time-Bounded Sequential Parameter Optimisation (TB-SPO)* algorithm by Hutter et al. [41] introduces a number of further modifications to the SMBO framework underlying the previously described SPO variants. In particular, in contrast to all SMBO procedures discussed so far, TB-SPO does not construct its initial model based on a large set of samples determined using a Latin hypercube design, but rather interleaves response measurements at randomly chosen points with ones taken at points that appear to be promising based on the current model. The initial model is based on a single sample only; when used for algorithm configuration, where the black-box function to be optimised represents the output of a parameterised algorithm, the default configuration for the algorithm is used as the design point at which this initial sample is taken. At any stage of the iterative model-based

search process that follows, response values are sampled at a series of design points in which odd-numbered points are determined by optimising an expected improvement measure (as is done in SPO$^+$), while even-numbered points are sampled uniformly at random from the given design space. (Mechanisms that achieve a different balance between promising and randomly chosen design points could lead to better performance but have not been explored so far.)

The number of design points at which response values are sampled between any two updates to the model is determined based on the time t required for constructing a new model and the search for promising parameter settings; to be precise, after at least two design points have been evaluated, further points are considered until the time used for evaluating design points since the last model update exceeds a user-defined multiple (or fraction) of the overhead t.

Finally, in order to reduce the computational overhead incurred by the model construction process, TB-SPO uses an approximate version of the standard Gaussian process models found in the other SPO variants. This so-called *projected process (PP) approximation* is based on the idea of representing explicitly only a randomly sampled subset of the given data points (here: pairs of input and response values) when building the Gaussian process model; if this subset comprises s data points, while the complete set has n data points, the time complexity of fitting a GP model decreases from $O(n^3)$ to $O((s+n) \cdot s^2)$, while the time required for predicting a response value (mean and variance of the predictive distribution at a given design point) decreases from $O(n^2)$ to $O(s^2)$ [56]. In the context of an SMBO procedure, this will typically lead to substantial savings, since the number of data points available increases over time, and n can easily reach values of several thousand, while effective PP approximations can be based on constant-size subsets with s no larger than 300 [41]. (Details on other, minor differences between TB-SPO and SPO$^+$ can be found in [41].)

3.4.4 Applications

As mentioned earlier, sequential model-based optimisation methods have primarily been developed for the optimisation of black-box functions, but can obviously be applied to algorithm configuration problems by defining the function to be optimised to be the performance of a target observed algorithm applied to one or more benchmark instances. The design points are thus algorithm configurations, and the response values capture the performance of A on the given benchmark instance(s) according to some performance measure m. In principle, SMBO procedures like those described earlier in this section can be applied to optimise a target algorithm on a set I of benchmark instances by using a measure m that captures the performance on the entire set I; however, as discussed earlier for the case of BasicILS, this tends to quickly become impractical as the size of I grows. Therefore, the empirical evaluation of SMBO procedures tends to be focussed on performance optimisation on single benchmark instances. Furthermore, because of the nature of the

response surface models used, SMBO methods are usually restricted to dealing with real- and integer-valued target algorithm parameters (although, very recently, [43] has introduced techniques that can handle categorical parameters).

Following an example from Bartz-Beielstein et al. [9], the SPO variants discussed in this section have been empirically evaluated using CMA-ES [25, 26, 27] – one of the best-performing gradient-free numerical optimisation procedures currently known – on several standard benchmark functions from the literature on gradient-free numerical optimisation (see, e.g., 26). The configuration space considered in these examples, which involve the convex Sphere function as well as the non-convex Ackley, Griewank and Rastrigin functions, is spanned by three real- and one integer-valued parameters of CMA-ES, and the performance measure was solution quality, achieved after a fixed number of evaluations of the respective benchmark function. The empirical results reported by Hutter et al. [38] for CMA-ES applied to the ten-dimensional instances of these functions indicate that SPO$^+$ tends to perform significantly better than SPO 0.3 and 0.4, which in turn appear to perform substantially better than SKO. In addition, Hutter et al. [38] considered the minimisation of the median number of search steps required by SAPS [33], a well-known stochastic local search algorithm for SAT, to solve a single benchmark instance obtained from encoding a widely studied quasi-group completion problem into SAT; in this case, four continuous parameters were optimised. The results from that experiment confirmed that SPO$^+$ tends to perform better than previous SPO variants and suggest that, at least on some configuration problems with a relatively modest number of predominently real-valued parameters, it can also yield slightly better results than FocusedILS when allowed the same number of target algorithm runs.

TB-SPO has been empirically compared to SPO$^+$ and FocusedILS on relatively simple algorithm configuration tasks involving the well-known SAT solver SAPS [33], with four continuous parameters, running on single SAT instances. In these experiments, TB-SPO was shown to perform significantly better than SPO$^+$ (sometimes achieving over 250-fold speedups), and moderately better than FocusedILS [41]. However, it is important to keep in mind that, unlike TB-SPO (and all other SMBO procedures covered in this section), FocusedILS explicitly deals with multiple problem instances, and can therefore be expected to perform substantially better on realistic algorithm configuration tasks. Furthermore, while SMBO procedures like TB-SPO do not require continuous algorithm parameters to be discretised, they presently cannot deal with conditional parameters, which are routinely encountered in the more challenging algorithm configuration tasks on which FocusedILS has been shown to be quite effective.

3.5 Other Approaches

In addition to the methods covered in the previous sections, there are many other procedures described in the literature that can, at least in principle, be applied to the algorithm configuration problem considered here.

Experimental design methods, such as full or fractional factorial designs, stratified random sampling, Latin hypercube designs and various other types of space-filling and uniform designs (see, e.g., 60), are applicable to algorithm configuration, but per se do not take into account one fundamental aspect of the problem: Namely, that we are interested in performance on multiple instances and have control over the number of instances used for evaluating any given configuration. Furthermore, when minimizing the run-time of the given target algorithm (a very common performance objective in algorithm configuration and parameter tuning), it is typically necessary to work with censored data from incomplete runs, and it can be beneficial to cut off runs early (as done by the adaptive capping strategy explained in Section 3.3.4). Perhaps more importantly, experimental design methods lack the heuristic guidance that is often crucial for searching large configuration spaces effectively.

Nevertheless, these simple design methods are sometimes used for the initialisation of more complex procedures (see Section 3.4 and [1], which will be discussed in slightly more detail later). There is also some evidence that in certain cases a method as simple as uniform random sampling, when augmented with adaptive capping or with the mechanism used by TB-SPO for evaluating configurations, can be quite effective (see the recent work [41]).

In principle, gradient-free numerical optimisation methods are directly applicable to parameter tuning problems, provided that all parameters are real-valued (and that there are no parameter dependencies, such as conditional parameters). Prominent and relatively recent methods that appear to be particularly suitable in this context are the covariance matrix adaptation evolution strategy (CMA-ES) by Hansen and Ostermeier [27] and the mesh adaptive direct search (MADS) algorithms by Audet and Orban [4]. Similarly, it is possible to use gradient-based numerical optimisation procedures – in particular, quasi-Newton methods such as the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm (see, e.g., 51) – in conjunction with suitable methods for estimating or approximating gradient information. However, in order to be applied in the context of configuring target algorithms with categorical and conditional parameters, these methods would require non-obvious modifications; we also expect that in order to be reasonably effective, they would need to be augmented with mechanisms for dealing with multiple problem instances and capped (or censored) runs. The same holds for standard methods for solving stochastic optimisation problems (see, e.g., 63).

The CALIBRA algorithm by Adenso-Diaz and Laguna [1], on the other hand, has been specifically designed for parameter tuning tasks. It uses a specific type of fractional factorial design from the well-known Taguchi methodology in combination with multiple runs of a local search procedure that gradually refines the region of interest. Unfortunately, CALIBRA can handle no more than five parameters, all of which need to be ordinal (the limitation to five parameters stems from the specific fractional designs used at the core of the procedure).

The CLASS system by Fukunaga [18, 19] is based on a genetic programming approach; it has been specifically built for searching a potentially unbounded space of the heuristic variable selection method used in an SLS-based SAT solver. Like

most genetic programming approaches, CLASS closely links the specification of the configuration space and the evolutionary algorithm used for exploring this space.

The Composer system developed by Gratch and Dejong [23] is based on an iterative improvement procedure not unlike that used in ParamILS; this procedure is conceptually related to racing techniques in that it moves to a new configuration only after gathering sufficient statistical evidence to conclude that this new configuration performs significantly better than the current one. In a prominent application, Gratch and Chien [22] used the Composer system to optimise five parameters of an algorithm for scheduling communication between a spacecraft and a set of ground-based antennas.

Ansótegui et al. [2] recently developed a gender-based genetic algorithm for solving algorithm configuration problems. Their GGA procedure supports categorical, ordinal and real-valued parameters; it also allows its user to express independencies between parameter effects by means of so-called variable trees – a concept that could be of particular interest in the context of algorithm configuration problems where such independencies are known by construction, or heuristic methods are are available for detecting (approximate) independencies. Although there is some evidence that GGA can solve some moderately difficult configuration problems more effectively than FocusedILS without capping [2], it appears to be unable to reach the performance of FocusedILS version 2.3 with aggressive capping on the most challenging configurations problems [40]. Unfortunately, GGA also offers less flexibility than FocusedILS in terms of the performance metric to be optimised. More algorithm configuration procedures based on evolutionary algorithms are covered in Chapter 2 of this book.

Finally, work originating from the Ph.D. project of Hutter [32] has recently overcome two major limitations of the sequential model-based optimisation methods discussed in Section 3.4 of this chapter by introducing a procedure that can handle categorical parameters while explicitly exploiting the fact that performance is evaluated on a set of problem instances. There is some evidence that this procedure, dubbed *Sequential Model-based Algorithm Configuration (SMAC)*, can, at least on some challenging configuration benchmarks, reach and sometimes exceed the performance of FocusedILS [43], and we are convinced that, at least in cases where the parameter response of a given target algorithm is reasonably regular and performance evaluations are very costly, such advanced SMBO methods hold great promise.

3.6 Conclusions and Future Work

Automated algorithm configuration and parameter tuning methods have been developed and used for more than a decade, and many of the fundamental techniques date back even further. However, it has only recently become possible to effectively solve complex configuration problems involving target algorithms with dozens of parameters, which are often categorical and conditional. This success is based in

part on the increased availability of computational resources, but has mostly been enabled by methodological advances underlying recent configuration procedures.

Still, we see much room (and, indeed, need) for future work on automated algorithm configuration and parameter tuning methods. We believe that in developing such methods, the fundamental features underlying all three types of methods discussed in this chapter can play an important role, and that the best methods will employ combinations of these. We further believe that different configuration procedures will likely be most effective for solving different types of configuration problems (depending, in particular, on the number and type of target algorithm parameters, but also on regularities in the parameter response). Therefore, we see a need for research aiming to determine which configurator is most effective under which circumstances. In fact, we expect to find situations in which the sequential or iterative application of more than one configuration procedure turns out to be effective – for example, one could imagine applying FocusedILS to find promising configurations in vast, discrete configuration spaces, followed by a gradient-free numerical optimisation method, such as CMA-ES, for fine-tuning a small number of real-valued parameters.

Overall, we believe that algorithm configuration techniques, such as the ones discussed in this chapter, will play an increasingly crucial role in the development, evaluation and use of state-of-the-art algorithms for challenging computational problems, where the challenge could arise from high computational complexity (in particular, $\mathcal{NP}$-hardness) or from tight resource constraints (e.g., in real-time applications). Therefore, we see great value in the design and development of software frameworks that support the real-world application of various algorithm configuration and parameter tuning procedures. The High-Performance Algorithm Lab (HAL), recently introduced by Nell et al. [50], is a software environment designed to support a wide range of empirical analysis and design tasks encountered during the development, evaluation and application of high-performance algorithms for challenging computational problems, including algorithm configuration and parameter tuning. Environments such as HAL not only facilitate the application of automated algorithm configuration and parameter tuning procedures, but also their development, efficient implementation and empirical evaluation.

In closing, we note that the availability of powerful and effective algorithm configuration and parameter tuning procedures has a number of interesting consequences for the way in which high-performance algorithms are designed and used in practice. Firstly, for developers and end users, it is now possible to automatically optimise the performance of (highly) parameterised solvers specifically for certain classes of problem instances, leading to potentially much improved performance in real-world applications. Secondly, while on-line algorithm control mechanisms that adjust parameter settings during the run of a solver (as covered, for example, in Chapters 6, 7 and 8 of this book) can in principle lead to better performance than the (static) algorithm configuration procedures considered in this chapter, we expect these latter procedures to be very useful in the context of (statically) configuring the parameters and heuristic components that determine the behaviour of the on-line control mechanisms. Finally, during algorithm development, it is no longer neces-

sary (or even desirable) to eliminate parameters and similar degrees of freedom, but instead, developers can focus more on developing ideas for realising certain heuristic mechanisms or components, while the precise instantiation can be left to automated configuration procedures [28]. We strongly believe that this last effect will lead to a fundamentally different and substantially more effective way of designing and implementing high-performance solvers for challenging computational problems.

Acknowledgements This chapter surveys and discusses to a large extent work carried out by my research group at UBC, primarily involving Frank Hutter, Kevin Leyton-Brown and Kevin Murphy, as well as Thomas Stützle at Université Libre de Bruxelles, to all of whom I am deeply grateful for their fruitful and ongoing collaboration. I gratefully acknowledge valuable comments by Frank Hutter, Thomas Stützle and Maverick Chan on earlier drafts of this chapter, and I thank the members of my research group for the many intellectually stimulating discussions that provide a fertile ground for much of our work on automated algorithm configuration and other topics in empirical algorithmics.

References

[1] Adenso-Diaz, B., Laguna, M.: Fine-tuning of algorithms using fractional experimental design and local search. Operations Research 54(1):99–114 (2006)
[2] Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP 2009), pp. 142–157 (2009)
[3] Applegate, D. L., Bixby, R. E., Chvátal, V., Cook, W. J.: The Traveling Salesman Problem: A Computational Study. Princeton University Press (2006)
[4] Audet, C., Orban, D.: Finding optimal algorithmic parameters using the mesh adaptive direct search algorithm. SIAM Journal on Optimization 17(3):642–664 (2006)
[5] Balaprakash, P., Birattari, M., Stützle, T.: Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement. In: Bartz-Beielstein, T., Blesa, M., Blum, C., Naujoks, B., Roli, A., Rudolph, G., Sampels, M. (eds) 4th International Workshop on Hybrid Metaheuristics, Proceedings, HM 2007, Springer Verlag, Berlin, Germany, Lecture Notes in Computer Science, vol. 4771, pp. 108–122 (2007)
[6] Balaprakash, P., Birattari, M., Stützle, T., Dorigo, M.: Estimation-based metaheuristics for the probabilistic traveling salesman problem. Computers & OR 37(11):1939–1951 (2010)
[7] Bartz-Beielstein, T.: Experimental Research in Evolutionary Computation: The New Experimentalism. Natural Computing Series, Springer Verlag, Berlin, Germany (2006)
[8] Bartz-Beielstein, T., Lasarczyk, C., Preuß, M.: Sequential parameter optimization. In: McKay, B., et al. (eds) Proceedings 2005 Congress on Evolutionary

Computation (CEC'05), Edinburgh, Scotland, IEEE Press, vol. 1, pp. 773–780 (2005)

[9] Bartz-Beielstein, T., Lasarczyk, C., Preuss, M.: Sequential parameter optimization toolbox, manual version 0.5, September 2008, available at http://www.gm.fh-koeln.de/imperia/md/content/personen/lehrende/bartz_beielstein_thomas/spotdoc.pdf (2008)

[10] Battiti, R., Brunato, M., Mascia, F.: Reactive Search and Intelligent Optimization. Operations Research/Computer Science Interfaces, Springer Verlag (2008)

[11] Birattari, M., Stützle, T., Paquete, L., Varrentrapp,K.: A racing algorithm for configuring metaheuristics. In: GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 11–18 (2002)

[12] Birattari, M., Yuan, Z., Balaprakash, P., Stützle, T.: F-race and Iterated F-Race: An overview. In: Bartz-Beielstein, T., Chiarandini, M., Paquete, L., Preuss, M. (eds) Experimental Methods for the Analysis of Optimization Algorithms, Springer, Berlin, Germany, pp. 311–336 (2010)

[13] Bűrmen, Á., Puhan, J., Tuma, T.: Grid restrained Nelder-Mead algorithm. Computational Optimization and Applications 34(3):359–375 (2006)

[14] Carchrae, T., Beck, J.: Applying machine learning to low knowledge control of optimization algorithms. Computational Intelligence 21(4):373–387 (2005)

[15] Chiarandini, M., Fawcett, C., Hoos, H.: A modular multiphase heuristic solver for post enrollment course timetabling (extended abstract). In: Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling PATAT (2008)

[16] Da Costa, L., Fialho, Á., Schoenauer, M., Sebag, M.: Adaptive Operator Selection with Dynamic Multi-Armed Bandits. In: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO'08), pp. 913–920 (2008)

[17] Fawcett, C., Hoos, H., Chiarandini, M.: An automatically configured modular algorithm for post enrollment course timetabling. Tech. Rep. TR-2009-15, University of British Columbia, Department of Computer Science (2009)

[18] Fukunaga, A. S.: Automated discovery of composite SAT variable-selection heuristics. In: Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02), pp. 641–648 (2002)

[19] Fukunaga, A. S.: Evolving local search heuristics for SAT using genetic programming. In: Genetic and Evolutionary Computation – GECCO-2004, Part II, Springer-Verlag, Seattle, WA, USA, Lecture Notes in Computer Science, vol. 3103, pp. 483–494 (2004)

[20] Gagliolo, M., Schmidhuber, J.: Dynamic algorithm portfolios. In: Amato, C., Bernstein, D., Zilberstein, S. (eds) Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics (AI-MATH-06) (2006)

[21] Gomes, C. P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning 24(1-2):67–100 (2000)

[22] Gratch, J., Chien, S. A.: Adaptive problem-solving for large-scale scheduling problems: A case study. Journal of Artificial Intelligence Research 4:365–396 (1996)
[23] Gratch, J., Dejong, G.: Composer: A probabilistic solution to the utility problem in speed-up learning. In: Rosenbloom, P., Szolovits, P. (eds) Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92), AAAI Press / The MIT Press, Menlo Park, CA, USA, pp. 235–240 (1992)
[24] Guerri, A., Milano, M.: Learning techniques for automatic algorithm portfolio selection. In: Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004), pp. 475–479 (2004)
[25] Hansen, N.: The CMA evolution strategy: A comparing review. In: Lozano, J., Larranaga, P., Inza, I., Bengoetxea, E. (eds) Towards a new evolutionary computation. Advances on estimation of distribution algorithms, Springer, pp. 75–102 (2006)
[26] Hansen, N., Kern, S.: Evaluating the CMA evolution strategy on multimodal test functions. In: Yao, X., et al. (eds) Parallel Problem Solving from Nature PPSN VIII, Springer, LNCS, vol. 3242, pp. 282–291 (2004)
[27] Hansen, N., Ostermeier, A.: Completely derandomized self-adaptation in evolution strategies. Evolutionary Computation 9(2):159–195 (2001)
[28] Hoos, H.: Computer-aided design of high-performance algorithms. Tech. Rep. TR-2008-16, University of British Columbia, Department of Computer Science (2008)
[29] Hoos, H., Stützle, T.: Local search algorithms for SAT: An empirical evaluation. Journal of Automated Reasoning 24(4):421–481 (2000)
[30] Hoos, H., Stützle, T.: Stochastic Local Search—Foundations and Applications. Morgan Kaufmann Publishers, USA (2004)
[31] Huang, D., Allen, T. T., Notz, W. I., Zeng, N.: Global optimization of stochastic black-box systems via sequential kriging meta-models. Journal of Global Optimization 34(3):441–466 (2006)
[32] Hutter, F.: Automated configuration of algorithms for solving hard computational problems. Ph.D. thesis, University of British Columbia, Department of Computer Science, Vancouver, BC, Canada (2009)
[33] Hutter, F., Tompkins, D. A., Hoos, H.: Scaling and Probabilistic Smoothing: Efficient Dynamic Local Search for SAT. In: Principles and Practice of Constraint Programming – CP 2002, Springer-Verlag, LNCS, vol. 2470, pp. 233–248 (2002)
[34] Hutter, F., Hamadi, Y., Hoos, H., Leyton-Brown, K.: Performance prediction and automated tuning of randomized and parametric algorithms. In: Principles and Practice of Constraint Programming – CP 2006, Springer-Verlag, LNCS, vol. 4204, pp. 213–228 (2006)
[35] Hutter F., Babić, D., Hoos, H., Hu, A. J.: Boosting verification by automatic tuning of decision procedures. In: Proc. Formal Methods in Computer-Aided Design (FMCAD'07), IEEE Computer Society Press, pp. 27–34 (2007)

[36] Hutter, F., Hoos, H., Stützle, T.: Automatic algorithm configuration based on local search. In: Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI-07), pp. 1152–1157 (2007)

[37] Hutter, F., Hoos, H., Leyton-Brown, K., Murphy, K.: An experimental investigation of model-based parameter optimisation: SPO and beyond. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO'09), ACM, pp. 271–278 (2009)

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

[39] Hutter, F., Hoos, H., Leyton-Brown, K., Stützle, T.: ParamILS: An automatic algorithm configuration framework (extended version). Tech. Rep. TR-2009-01, University of British Columbia, Department of Computer Science (2009)

[40] Hutter, F., Hoos, H., Leyton-Brown K.: Sequential model-based optimization for general algorithm configuration (extended version). Tech. Rep. TR-2010-10, University of British Columbia, Department of Computer Science (2010)

[41] Hutter, F., Hoos, H., Leyton-Brown, K., Murphy, K.: Time-bounded sequential parameter optimization. In: Proceedings of the 4th International Conference on Learning and Intelligent Optimization (LION 4), Springer-Verlag, LNCS, vol. 6073, pp. 281–298 (2010)

[42] Hutter, F., Hoos, H., Leyton-Brown, K.: Automated configuration of mixed integer programming solvers. In: Proceedings of the 7th International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2010), Springer-Verlag, LNCS, vol. 6140, pp. 186–202 (2010)

[43] Hutter, F., Hoos, H., Leyton-Brown, K.: Extending sequential model-based optimization to general algorithm configuration. To appear in: *Proceedings of the 5th International Conference on Learning and Intelligent Optimization (LION 5)* (2011)

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

[45] KhudaBukhsh, A., Xu, L., Hoos, H., Leyton-Brown, K.: SATenstein: Automatically building local search SAT solvers from components. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09), pp 517–524 (2009)

[46] Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., Shoham, Y.: A portfolio approach to algorithm selection. In: Rossi, F. (ed) Principles and Practice of Constraint Programming – CP 2003, Springer Verlag, Berlin, Germany, Lecture Notes in Computer Science, vol. 2833, pp. 899–903 (2003)

[47] Lourenço, H. R., Martin, O., Stützle, T.: Iterated local search. In: Glover, F., Kochenberger, G. (eds) Handbook of Metaheuristics, Kluwer Academic Publishers, Norwell, MA, USA, pp. 321–353 (2002)

[48] Maron, O., Moore, A. W.: Hoeffding races: Accelerating model selection search for classification and function approximation. In: Advances in neural information processing systems 6, Morgan Kaufmann, pp. 59–66 (1994)
[49] Nelder, J. A., Mead, R.: A simplex method for function minimization. The Computer Journal 7(4):308–313 (1965)
[50] Nell, C. W., Fawcett, C., Hoos, H., Leyton-Brown K.: HAL: A framework for the automated design and analysis of high-performance algorithms. To appear in: *Proceedings of the 5th International Conference on Learning and Intelligent Optimization (LION 5)* (2011)
[51] Nocedal, J., Wright, S. J.: Numerical Optimization, 2nd edn. Springer-Verlag (2006)
[52] Nouyan, S., Campo, A., Dorigo, M.: Path formation in a robot swarm: Self-organized strategies to find your way home. Swarm Intelligence 2(1):1–23 (2008)
[53] Pellegrini, P., Birattari, M.: The relevance of tuning the parameters of metaheuristics. a case study: The vehicle routing problem with stochastic demand. Tech. Rep. TR/IRIDIA/2006-008, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium (2006)
[54] Pop, M., Salzberg, S. L., Shumway, M.: Genome sequence assembly: Algorithms and issues. Computer 35(7):47–54 (2002)
[55] Prasad, M. R., Biere, A., Gupta, A.: A survey of recent advances in SAT-based formal verification. International Journal on Software Tools for Technology Transfer 7(2):156–173 (2005)
[56] Rasmussen, C. E., Williams, C. K. I.: Gaussian Processes for Machine Learning. The MIT Press (2006)
[57] Rice, J.: The algorithm selection problem. Advances in Computers 15:65–118 (1976)
[58] Rossi-Doria, O., Sampels, M., Birattari, M., Chiarandini, M., Dorigo, M., Gambardella, L. M., Knowles, J. D., Manfrin, M., Mastrolilli, M., Paechter, B., Paquete, L., Stützle, T.: A comparison of the performance of different metaheuristics on the timetabling problem. In: Burke, E. K., Causmaecker, P. D. (eds) Practice and Theory of Automated Timetabling IV, 4th International Conference, PATAT 2002, Selected Revised Papers, Springer, Lecture Notes in Computer Science, vol. 2740, pp. 329–354 (2003)
[59] Sacks, J., Welch, W., Mitchell, T., Wynn, H.: Design and analysis of computer experiments (with discussion). Statistical Science 4:409–435 (1989)
[60] Santner, T., Williams, B., Notz, W.: The Design and Analysis of Computer Experiments. Springer Verlag, New York (2003)
[61] Schiavinotto, T., Stützle, T.: The linear ordering problem: Instances, search space analysis and algorithms. Journal of Mathematical Modelling and Algorithms 3(4):367–402 (2004)
[62] Schonlau, M., Welch, W. J., Jones, D. R.: Global versus local search in constrained optimization of computer models. In: Flournoy, N., Rosenberger, W., Wong, W. (eds) New Developments and Applications in Experimental Design,

vol. 34, Institute of Mathematical Statistics, Hayward, California, pp. 11–25 (1998)

[63] Spall, J.: Introduction to Stochastic Search and Optimization. John Wiley & Sons, Inc., New York, NY, USA (2003)

[64] Stützle, T., Hoos, H.: MAX-MIN Ant System. Future Generation Computer Systems 16(8):889–914 (2000)

[65] Thachuk, C., Shmygelska, A., Hoos, H.: A replica exchange Monte Carlo algorithm for protein folding in the hp model. BMC Bioinformatics 8(342) (2007)

[66] Tompkins, D., Hoos, H.: Dynamic Scoring Functions with Variable Expressions: New SLS Methods for Solving SAT. In: Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT 2010), Springer-Verlag, LNCS, vol. 6175, pp. 278–292 (2010)

[67] Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla-07: The design and analysis of an algorithm portfolio for SAT. In: Principles and Practice of Constraint Programming – CP 2007, Springer Berlin / Heidelberg, LNCS, vol. 4741, pp. 712–727 (2007)

[68] Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research 32:565–606 (2008)

[69] Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla2009: An Automatic Algorithm Portfolio for SAT, Solver Description, SAT Competition 2009 (2009)

[70] Xu, L., Hoos, H., Leyton-Brown, K.: Hydra: Automatically configuring algorithms for portfolio-based selection. In: Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10), pp. 210–216 (2010)

[71] Yuan, Z., Fügenschuh, A., Homfeld, H., Balaprakash, P., Stützle T., Schoch M.: Iterated greedy algorithms for a real-world cyclic train scheduling problem. In: Blesa, M., Blum, C., Cotta, C., Fernández, A., Gallardo, J., Roli, A., Sampels, M. (eds) Hybrid Metaheuristics, Lecture Notes in Computer Science, vol. 5296, Springer Berlin / Heidelberg, pp. 102–116 (2008)