ParamILS: An Automatic Algorithm Configuration Framework

Frank Hutter HUTTER@CS.UBC.CA
Holger H. Hoos HOOS@CS.UBC.CA
Kevin Leyton-Brown KEVINLB@CS.UBC.CA
University of British Columbia, 2366 Main Mall
Vancouver, BC, V6T1Z4, Canada

Thomas Stützle STUETZLE@ULB.AC.BE
Université Libre de Bruxelles, CoDE, IRIDIA
Av. F. Roosevelt 50 B-1050 Brussels, Belgium

Abstract

The identification of performance-optimizing parameter settings is an important part of the development and application of parameterized algorithms. We propose an automatic algorithm configuration framework in which the settings of discrete parameters are optimized to yield maximal performance of a target algorithm for a given class of problem instances. We begin with a thorough experimental analysis of this algorithm configuration problem, using a simple random search procedure to explore the parameter configuration spaces of high-performance algorithms for SAT and mixed integer programming. Next, we present a family of local-search-based algorithm configuration procedures, along with adaptive capping techniques for accelerating them. Finally, we describe the results of a comprehensive experimental evaluation of our methods, based on the configuration of prominent complete and incomplete algorithms for SAT, and the commercial software CPLEX for mixed integer programming. In all cases, these algorithms' default parameter settings had previously been manually optimized with considerable effort; nevertheless, using our automated algorithm configuration procedures, we achieved substantial performance improvements.

1. Introduction

Many high-performance algorithms have parameters whose settings control important aspects of their behaviour. This is particularly the case for the heuristic procedures used for solving computationally hard problems, for example in AI, where the problem of setting an algorithm's free parameters in order to optimize its performance on a class of problem instances is ubiquitous. As an example, consider CPLEX, a commercial solver for mixed integer programming problems.[1] CPLEX version 10.1.1 has 159 user-specifiable parameters, about 80 of which affect the solver's search mechanism and can be configured to improve performance. Anecdotal evidence suggests that many man-years have been devoted to establishing default settings of these parameters that allow CPLEX to robustly achieve high performance on a wide range of mixed integer programming problems.

It is widely acknowledged that finding performance-optimizing parameter configurations of heuristic algorithms often requires considerable effort (see, e.g., Gratch and Chien, 1996; Johnson, 2002; Diao et al., 2003; Birattari, 2005; Adenso-Diaz and Laguna, 2006), and in many cases, this tedious task is performed manually in an ad-hoc way. Automating this important task is of high practical relevance in several contexts:

1. http://www.ilog.com/products/cplex/

- **Development of complex algorithms:** Setting the parameters of a heuristic algorithm is a highly labour-intensive task, and indeed can consume a large fraction of overall development time. The use of automated algorithm configuration methods can lead to significant time savings and potentially achieve better results than manual, ad-hoc methods.

- **Empirical studies, evaluations, and comparisons of algorithms:** A central question in comparing heuristic algorithms is whether one algorithm outperforms another because it is fundamentally superior, or because its developers more successfully optimized its parameters (Johnson, 2002). Automatic algorithm configuration methods can mitigate this problem of unfair tuning and thus facilitate more meaningful comparative studies.

- **Practical use of algorithms.** The ability of complex heuristic algorithms to solve large and hard problem instances often depends critically on the use of suitable parameter settings. End users often have little or no knowledge about an algorithm's parameter configuration space and thus simply use the default settings. Even if it has been carefully optimized on a standard benchmark set, such a default configuration may not perform well on the particular problem instances encountered by a user. Automatic algorithm configuration methods can be used to improve performance in a principled and convenient way.

Various research communities have developed strategies for automatic algorithm configuration. Briefly, these include exhaustive enumeration, hill-climbing (Gratch and Dejong, 1992), beam search (Minton, 1993), genetic programming (Oltean, 2005), experimental design approaches (Coy et al., 2001), sequential parameter optimization (Bartz-Beielstein, 2006), optimization of parameters one at a time (den Besten et al., 2001), racing algorithms (Birattari et al., 2002; Birattari, 2004, 2005; Balaprakash et al., 2007), and combinations of fractional experimental design and local search (Adenso-Diaz and Laguna, 2006).

We discuss this and other related work more extensively in Section 10. Here, we note that while some other authors refer to the optimization of an algorithm's performance by setting its (typically few and numerical) parameters as *parameter tuning*, we use the term *algorithm configuration* (or simply, *configuration*). This is motivated by the fact that we are interested in methods that can deal with a potentially large number of parameters, each of which can be numerical, ordinal (e.g., low, medium, or high) or categorical (e.g., choice of heuristic). Categorical parameters can be used to select and combine discrete building blocks of an algorithm (e.g., preprocessing and variable ordering heuristics); consequently, our general view of algorithm configuration includes the automated construction of a heuristic algorithm from such building blocks.

To avoid potential confusion between algorithms whose performance is optimized and algorithms used for carrying out this optimization task, we refer to the former as *target algorithms* and the latter as *configuration procedures* (or simply *configurators*). This setup is illustrated in Figure 1. Different algorithm configuration problems have been considered in the literature, including setting parameters on a per-instance basis and adapting the parameters while the algorithm is running; we defer a discussion of these approaches to Section 10.

In recent work, we introduced ParamILS, a versatile iterated local search (ILS) approach for automated algorithm configuration, and two instantiations of the framework, BasicILS and FocusedILS (Hutter et al., 2007b). We showed that these configuration procedures efficiently optimized the performance of several high-performance heuristic tree search and local search algorithms for the propositional satisfiability (SAT) and most probable explanation (MPE) problems.

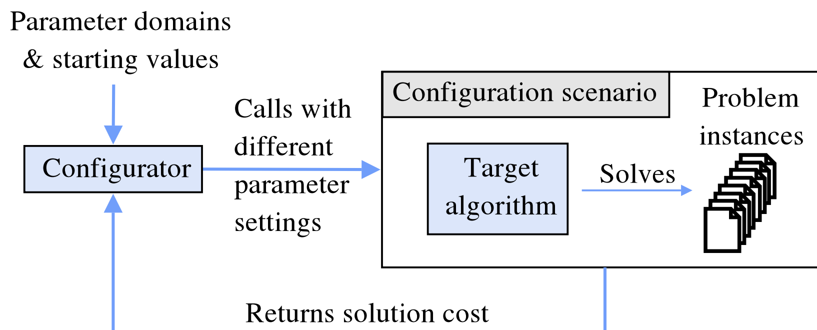


Figure 1: A configuration scenario includes an algorithm to be configured and a collection of instances. A configuration procedure executes the target algorithm with specified parameter settings on some or all of the instances, receives information about the performance of these runs, and uses this information to decide about what subsequent parameter configurations to evaluate.

In a subsequent application study, we gave further evidence of ParamILS's effectiveness by configuring a highly optimized SAT solver for industrially relevant software and hardware verification instances, resulting in performance improvements of several orders of magnitude and a significant enhancement in the state of the art in solving these problems (Hutter et al., 2007a).

Here, we extend and expand upon our earlier work in several important ways.

1. After introducing in more detail the algorithm configuration problem we are interested in, we provide additional insights into its properties. In particular, in Section 4, we study a number of application scenarios in a manner that is independent of the particular configuration procedure being used. Specifically, we investigate performance variation across parameter configurations, hardness variation across instances, configurator "overconfidence" (inability to generalize from observed to unobserved problem instances), and the impact of varying the amount of benchmark data and the per-run cutoff time on the performance of the configuration procedure.

2. After an expanded discussion of the ParamILS framework and its two instantiations, BasicILS and FocusedILS, we present additional evidence for the efficacy of these automatic algorithm configuration methods. In particular, in Section 5, we demonstrate that the iterated local search approach underlying ParamILS achieves better results than either random search or a simple local search procedure.

3. We introduce *adaptive capping*, a new technique that can be used to enhance search-based algorithm configuration procedures independently of the underlying search strategy (Section 7). Adaptive capping is based on the idea of avoiding unnecessary runs of the algorithm to be configured by developing bounds on the performance measure to be optimized. We present two instantiations of this technique and demonstrate that by using these, the performance of ParamILS can be substantially improved.

4. We present extensive evidence that ParamILS can find parameter configurations of complex and highly optimized algorithms that substantially outperform the algorithms' default settings (Section 8). In particular, we apply our automatic algorithm configuration procedures to the aforementioned commercial optimization tool CPLEX, one of the most powerful, widely used and complex optimization algorithms we are aware of. As stated in the CPLEX user manual (version 10.0, page 247),

   > "A great deal of algorithmic development effort has been devoted to establishing default ILOG CPLEX parameter settings that achieve good performance on a wide variety of MIP models."

   We demonstrate consistent improvements over this default parameter configuration for a wide range of practically relevant instance distributions; in some cases, we were able to speed up CPLEX by over an order of magnitude on average on previously unseen test instances. We also report experiments with new, challenging configuration scenarios for the SAT algorithms SAPS and SPEAR and demonstrate that we can achieve speedup factors in average-case performance of up to 3 000 (SAPS, graph colouring) and 80 (SPEAR, software verification).

5. In Section 9, we present evidence that ParamILS can be used to configure its own parameters—after all, it is itself a parameterized, heuristic search algorithm.

2. Problem Statement and Notation

The algorithm configuration problem we consider in this work can be informally stated as follows: given an algorithm, a set of parameters for the algorithm and a set of input data, find parameter values under which the algorithm achieves the best possible performance on the input data.

Now we define this problem more formally, and introduce notation that we will use throughout the paper. Let $\mathcal{A}$ denote an algorithm, and let $p_1, \ldots, p_k$ be parameters of $\mathcal{A}$. We denote the domain of possible values for each parameter p_i as Θ_i; these domains can be infinite (as for continuous and integer numerical parameters), finite and ordered (ordinal), or finite and unordered (categorical). Throughout this work, we assume that all parameter domains are finite. This assumption can be met by discretizing all numerical parameters to a finite number of options; furthermore, we do not exploit the canonical ordering relation for numerical parameters. In our experience, these limitations are not overly restrictive; they can, in principle, be overcome by extending our algorithm configuration procedures.

Our problem formulation allows us to express conditional parameter dependencies; for example, one algorithm parameter might be used to select among search heuristics, with each heuristic's behaviour controlled by further parameters. In this case, the values of these further parameters are irrelevant if the heuristic is not selected. ParamILS exploits this and effectively searches the space of equivalence classes in parameter configuration space. In addition, our formulation supports constraints on feasible combinations of parameter values. We use $\Theta \subseteq \Theta_1 \times \ldots \times \Theta_k$ to denote the space of all feasible parameter configurations, and $\mathcal{A}(\theta)$ to refer to algorithm $\mathcal{A}$ with parameter configuration $\theta \in \Theta$.

Let $\mathcal{D}$ denote a probability distribution over a space Π of problem instances, and let us denote an element of Π as π. Sometimes, $\mathcal{D}$ may be given in form of a random instance generator, or as a

distribution over instances from different generators. More commonly, Π consists of a finite sample of instances to which we have access. In that case, we define $\mathcal{D}$ as the uniform distribution over Π.

There are many ways of measuring an algorithm's performance. For example, we might be interested in minimizing computational resources consumed by the given algorithm (e.g., runtime, memory or communication bandwidth), or in maximizing the quality of the results produced. Since high-performance algorithms for computationally-challenging problems are often randomized, their behaviour can vary significantly between multiple runs. Thus, an algorithm will not always achieve the same performance, even when run repeatedly with fixed parameters on a single problem instance. Our overall goal must therefore be to choose parameter settings that minimize some cost statistic of the algorithm's performance across the input data. We denote this statistic as $c(\theta)$. For example, we might aim to minimize mean runtime or median solution cost.

With this intuition in mind, we now define the algorithm configuration problem formally.

Definition 1 (Algorithm Configuration Problem)**.** *An instance of the* algorithm configuration *problem is a 7-tuple* $\langle \mathcal{A}, \Theta, \mathcal{D}, \mathcal{S}, \kappa, o, m \rangle$*, where:*

- $\mathcal{A}$ *is a parameterised algorithm;*
- Θ *is the parameter configuration space of* $\mathcal{A}$*;*
- $\mathcal{D}$ *is a distribution over problem instances with domain* Π*;*
- $\mathcal{S}$ *is a distribution over allowed random number seeds (for deterministic algorithms* $\mathcal{A}$ *the domain of this distribution has cardinality one);*
- κ *is a cutoff time (or* captime*), after which each run of* $\mathcal{A}$ *will be terminated;*
- $o : \Theta, \mathcal{S}, \Pi, \mathbb{R} \rightarrow \mathbb{R}$ *is a function that measures the cost of running* $\mathcal{A}(\theta)$ *with seed* $s \in \mathcal{S}$ *on an instance* $\pi \in \Pi$ *with captime* $\kappa \in \mathbb{R}$ *(examples are runtime for solving the instance, or cost of the solution found within captime* κ*; in the former case,* o *must also define a cost for runs that do not complete within the captime); and*
- m *is a statistical population parameter. (Examples are expectation, median, and variance.)*

Any parameter configuration $\theta \in \Theta$ *is a candidate solution of the algorithm configuration problem. The* cost *of a candidate solution is*

$$c(\theta) = m_{\pi \sim \mathcal{D}, s \sim \mathcal{S}} \left[o(\theta, s, \pi, \kappa) \right], \tag{1}$$

the statistical population parameter to be optimized across instances from distribution $\mathcal{D}$ *and seeds from distribution* $\mathcal{S}$*. An optimal solution,* θ^**, minimises* $c(\theta)$*:*

$$\theta^* \in \arg\min_{\theta} c(\theta). \tag{2}$$

An *algorithm configuration procedure*, abbreviated as *configuration procedure* or *configurator*, is a procedure for tackling the algorithm configuration problem. Unfortunately, at least for the algorithm configuration problems considered in this article, we cannot optimize c directly, since this function cannot be written analytically. Instead, we must execute a sequence of runs $\mathbf{R}$ of the target algorithm $\mathcal{A}$ with different parameter configurations, derive empirical estimates of c's

values at particular points in configuration space, and use them to identify a configuration with low expected cost.

We denote the sequence of runs executed by a configurator as $\mathbf{R} = ((\theta_1, \pi_1, s_1, \kappa_1, o_1), \ldots, (\theta_n, \pi_n, s_n, \kappa_n, o_n))$. The ith run is described by five values. First, $\theta_i \in \Theta$ denotes the parameter configuration being evaluated. Second, $\pi_i \in \Pi$ denotes the instance on which the algorithm is run. Third, $s_i \in \mathcal{S}$ denotes the random number seed used in the run. Fourth, κ_i denotes the run's captime; when optimizing solution quality reached within a fixed time κ, we would always use $\kappa_i = \kappa$, but when optimizing functions of runtime, we are free to choose $\kappa_i \leq \kappa$. Finally, $o_i = o(\theta_i, s_i, \pi_i, \kappa_i)$ denotes the outcome of the run. This is the statistic of the run upon which c depends. Note that each of θ, π, s, κ, and o can vary from one element of $\mathbf{R}$ to the next, regardless of whether or not the other elements are held constant. Note also that $\mathbf{R}$ is not determined all at once, but is typically chosen in an online manner. We denote the ith run of $\mathbf{R}$ as $\mathbf{R}[i]$ and the subsequence of runs using parameter configuration θ (i.e., those runs with $\theta_i = \theta$) as $\mathbf{R}_\theta$. The configuration procedures considered in this article compute empirical estimates of $c(\theta)$ based solely on $\mathbf{R}_\theta$, but in principle other methods could be used.

In our algorithm configuration problems, the cost $c(\theta)$ of a parameter configuration θ is an unobservable quantity, and we need to estimate this quantity both during runtime of a configurator, and offline for evaluation purposes. For this purpose, we introduce the notion of a cost estimate.

Definition 2 (Cost Estimate)**.** *Given an algorithm configuration problem $\langle \mathcal{A}, \Theta, \mathcal{D}, \mathcal{S}, \kappa, o, m \rangle$, a* cost estimate $\hat{c}(\theta, \mathbf{R})$ *of a cost $c(\theta) = m_{\pi \sim \mathcal{D}, s \sim \mathcal{S}}\left[o(\theta, s, \pi, \kappa)\right]$ based on a sequence of runs $\mathbf{R} = ((\theta_1, \pi_1, s_1, \kappa_1, o_1), \ldots, (\theta_n, \pi_n, s_n, \kappa_n, o_n))$ is $m'(\{o_i \mid \theta_i = \theta\})$, where m' is the sample statistic analogue to the statistical population parameter m.*

For example, when $c(\theta)$ is mean runtime over a distribution of instances and random number seeds, $\hat{c}(\theta, \mathbf{R})$ is the sample mean runtime of runs $\mathbf{R}_\theta$.

All configuration procedures in this paper are anytime algorithms in the sense that at all times they keep track of the configuration currently believed to have the lowest cost; we refer to this configuration as the *incumbent configuration*, or short the *incumbent*, $\widehat{\theta^*}$. We evaluate a configurator's performance at time t by means of its incumbent's training and test performance, defined as follows.

Definition 3 (Training performance)**.** *When at some time t a configurator has performed a sequence of runs $\mathbf{R} = ((\theta_1, \pi_1, s_1, \kappa_1, o_1), \ldots, (\theta_n, \pi_n, s_n, \kappa_n, o_n))$ to solve an algorithm configuration problem $\langle \mathcal{A}, \Theta, \mathcal{D}, \mathcal{S}, \kappa, o, m \rangle$, and has thereby found incumbent configuration $\widehat{\theta^*}$, then its* training performance *at time t is defined as the cost estimate $\hat{c}(\widehat{\theta^*}, \mathbf{R})$.*

The set of instances $\{\pi_1, \ldots, \pi_n\}$ discussed above is called the *training set*. While the true cost of a parameter configuration cannot be computed exactly, it can be estimated using training performance. However, as we shall see later (in Section 4.3), training performance of a configurator is a biased estimator of its incumbent's true cost. In order to achieve unbiased estimates at least during offline evaluation, we set aside a fixed set of instances $\{\pi'_1, \ldots, \pi'_T\}$ (called the *test set*) and random seeds $\{s'_1, \ldots, s'_T\}$, both unknown to the configurator, and use these for evaluation.

Definition 4 (Test performance)**.** *At some time t, let a configurator's incumbent for an algorithm configuration problem $\langle \mathcal{A}, \Theta, \mathcal{D}, \mathcal{S}, \kappa, o, m \rangle$ be $\widehat{\theta^*}$ (this is found by means of executing a sequence of runs on the training set). Furthermore, let $\mathbf{R}' = ((\theta^*, \pi'_1, s'_1, \kappa, o_1), \ldots, (\theta^*, \pi'_T, s'_T, \kappa, o_T))$ be a sequence of runs on the T instances and random number seeds in the test set (which is performed*

offline for evaluation purposes), then the configurator's test performance *at time t is defined as the cost estimate $\hat{c}(\widehat{\theta}^*, \mathbf{R}')$.*

Throughout this article, we aim to minimize mean runtime. (See Section 3.1.1 for discussion of that choice.) Thus, a configurator's training performance is the mean runtime of the runs it performed with the incumbent. Its test performance is the mean runtime of the incumbent on the test set.

It is not obvious how an automatic algorithm configurator should choose runs in order to best minimize $c(\theta)$ within a given time budget. In effect, this issue is the topic of the rest of the paper. We note that it can be subdivided into the following three questions:

1. Which parameter configurations $\Theta' \subseteq \Theta$ should be evaluated?
2. Which problem instances $\Pi_{\theta'} \subseteq \Pi$ should be used for evaluating each $\theta' \in \Theta'$, and how many runs should be performed on each instance?
3. Which cutoff time κ should be used for each run?

We will demonstrate in Section 4 that each of these decisions plays an important role in constructing a high-performance procedure for solving the automatic algorithm configuration problem we consider in this work, and that the choices made on each of these issues can interact in interesting ways. Sections 5–7 deal with each of these questions in turn.

3. Experimental Preliminaries

In this section we give background information about the computational experiments presented in the following sections. First, we describe the design of our experiments. Second, we present the configuration scenarios (algorithm/benchmark data combinations) used throughout the paper. Finally, we describe the low-level details of our experimental setup.

3.1 Experimental Design

Here we describe our choice of objective function and the methods we used for selecting instances and seeds.

3.1.1 CONFIGURATION OBJECTIVE: MINIMUM MEAN RUNTIME

In Section 2 we mentioned that the algorithm configuration problem can be stated in terms of various different cost statistics. Indeed, in our past work we explored several of them: maximizing solution quality achieved in a given time, minimizing the runtime required to reach a given solution quality, and minimizing the runtime required to solve a single problem instance (Hutter et al., 2007b).

In this work we concentrate on only one objective: minimizing the mean runtime over instances from distribution $\mathcal{D}$. This optimization objective naturally occurs in many practical applications. It is also interesting theoretically, since it means that there is a strong correlation between $c(\theta)$ and the amount of time required to obtain a good empirical estimate of $c(\theta)$. In Section 7 we exploit this relationship to significantly speed up our configuration procedures. One might wonder whether mean is the right way to aggregate runtimes. In some preliminary experiments, we found that minimizing mean runtime led to parameter configurations with overall good runtime performance, including rather competitive median runtimes, while minimizing median runtime yielded less robust

parameter configurations that timed out on a large (but $< 50\%$) fraction of the benchmark instances. However, even when using the mean, we encounter runs that do not terminate within the given cutoff time; throughout this study, such unsuccessful runs are scored as if they terminated in ten times the cutoff.

3.1.2 SELECTING INSTANCES AND SEEDS

As mentioned above, often only a finite set Π of instances is available upon which to evaluate our algorithm. This is the case in the experiments we report here. Throughout the paper, all configuration experiments are performed on a training set containing half of the given benchmark instances, while the remaining instances are solely used as a test set for the purpose of evaluating the resulting parameter configuration. Since testing is an offline process, we always evaluate a parameter setting by running the target algorithm on every benchmark instance in the test set. However, during the configuration process (using the training set) efficiency matters, and hence we do not always evaluate a parameter setting on every benchmark instance. Thus, we are faced with the question: which instances should we use? Specifically, when we wish to estimate cost statistics $c(\theta)$ based on N runs, how should the instances for these N runs be chosen in order to obtain the most reliable estimates, particularly in order to support accurate comparisons of different parameter configurations?

Following common practice (see, e.g., Ridge and Kudenko, 2006), we blocked on instances; that is, we ensured that whenever two parameter configurations were compared, exactly the same instances were used to evaluate them. This avoided noise effects due to differences between instances (intuitively, we did not make the mistake of considering a configuration that happened to be tested on easier instances to be better than it really is). Similarly, for randomized algorithms, we blocked on random seeds.

Furthermore, when dealing with randomized target algorithms, there is a tradeoff between the number of problem instances used and the number of independent runs performed on each instance. In the extreme case, for a given sample size N, one could perform N runs on a single instance or a single run on each of N different instances. This latter strategy is known to result in minimal variance of the estimator for common optimization objectives such as minimization of mean runtime (which we consider in this study) or maximization of mean solution quality (Birattari, 2005). Consequently, we only performed multiple runs per instance if we wished to acquire more samples of the cost distribution than there were instances in the training set.

Based on these considerations, the configuration procedures we study in this article have been implemented to take a list of $\langle$instance, random number seed$\rangle$ pairs as one of their inputs. Empirical estimates $\hat{c}_N(\theta)$ of the cost statistic $c(\theta)$ to be optimized were determined from the first N $\langle$instance, seed$\rangle$ pairs in that list. Each list of $\langle$instance, seed$\rangle$ pairs was constructed as follows. Given a training set consisting of M problem instances, for $N \leq M$, we drew a sample of N instances uniformly at random and without replacement and added them to the list. If we wished to evaluate an algorithm on more samples than we had training instances, which could happen in the case of randomized algorithms, we repeatedly drew random samples of size M as described before, where each such batch corresponded to a random permutation of the N training instances, and added a final sample of size $N \bmod M < M$ as in the case $N \leq M$. As each sample was drawn, it was paired with a random number seed that was chosen uniformly at random from the set of all possible seeds and added to the list of $\langle$instance, seed$\rangle$ pairs.

Configuration scenario	Type of benchmark instances & citation
SAPS-SWGCP	Graph colouring (Gent et al., 1999)
SPEAR-SWGCP	Graph colouring (Gent et al., 1999)
SAPS-QCP	Quasigroup completion (Gomes and Selman, 1997)
SPEAR-QCP	Quasigroup completion (Gomes and Selman, 1997)
CPLEX-REGIONS100	Combinatorial Auctions (CATS) (Leyton-Brown et al., 2000)

Table 1: Overview of our five INDEPTH configuration scenarios.

3.1.3 COMPARISON OF CONFIGURATION PROCEDURES

Since the choice of instances (and to some degree of seeds) is very important for the final outcome of the optimization, in our experimental evaluations we always performed a number of independent repetitions of each configuration procedure (typically 25), and report the mean and standard deviation over those. We created a separate list of instances and seeds for each repetition as explained above, where the kth repetition of each configuration procedure uses the same kth list of instances and seeds. (Note, however, that the disjoint test set used to measure performance of parameter configurations is identical for all repetitions.)

When we performed a statistical test to compare two configuration procedures, we carried out 25 repetitions of each procedure, collecting their final results. Since the kth repetition of both procedures shared the same kth list of instances and seeds, we performed a *paired* statistical test in order to compare the two. In particular, we performed a two-sided paired Max-Wilcoxon test with the Null hypothesis that there was no difference in the performances; we consider p-values below 0.05 to be statistically significant. The p-values reported in all tables were derived using this test; p-values shown in parentheses refer to cases where the procedure we expected to perform better actually performed worse.

3.2 Configuration Scenarios

The problems studied experimentally in this paper can be divided into two classes. First, in most of the paper (Sections 4–7 and 9), we perform an in-depth analysis of five combinations of high-performance algorithms with widely-studied benchmark data. Table 1 gives an overview of these five scenarios, which we dub the INDEPTH scenarios. The algorithms and benchmark instance sets used in these scenarios are described in detail in Sections 3.2.1 and 3.2.2, respectively.

Second, in Section 8, we use the same target algorithms to tackle nine additional complex configuration scenarios, which we refer to as BROAD scenarios. These are described in detail in Section 8; an overview is shown in Table 12 on page 39.

3.2.1 TARGET ALGORITHMS

Our three target algorithms are listed in Table 2 along with their configurable parameters.

SAPS The first algorithm in our experiments is SAPS, a high-performance dynamic local search algorithm for SAT solving (Hutter et al., 2002); we use the UBCSAT implementation (Tompkins and Hoos, 2004). When introduced in 2002, SAPS was a state-of-the-art solver, and it still offers competitive performance on many instances. We chose to study this algorithm because it is well-known, it has relatively few parameters, and we are intimately familiar with it. The original defaults

Algorithm	Parameter type	# parameters of that type	# values considered for each
SAPS	Continuous	4	7
SPEAR	Categorical	10	2–20
	Integer	4	5–8
	Continuous	12	3–6
CPLEX	Categorical	50	2–7
	Integer	8	5–7
	Continuous	5	3–5

Table 2: Parameter overview for the algorithms we consider. More information on the parameters for each algorithm is given in the text. A detailed list of all parameters and the values we considered can be found in an online appendix at `http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/algorithms.html`.

for SAPS' four continuous parameters were set through manual configuration based on experiments with prominent benchmark instances. These four continuous parameters control the scaling and smoothing of clause weights, as well as the percentage of random steps. Having subsequently gained more experience with SAPS' parameters in previous work on parameter configuration (Hutter et al., 2006) for more general problem classes, we chose promising intervals for each parameter, including but not centered at the original default. We then picked seven possible values for each parameter spread uniformly across its respective interval, resulting in 2401 possible parameter configurations (these are exactly the same values as used in (Hutter et al., 2007b)). As the starting configuration for ParamILS, we used the center of each parameter's interval.

SPEAR The second algorithm we consider is SPEAR, a recent tree search algorithm for solving SAT problems. SPEAR is a state-of-the-art SAT solver for industrial instances, and with appropriate parameter settings it is the best available solver for certain types of hardware and software verification instances (Hutter et al., 2007a). (Indeed, the parameter configuration procedures we describe in this work played a crucial role in SPEAR's development (Hutter et al., 2007a); SPEAR subsequently won the quantifier-free bit-vector arithmetic category of the 2007 Satisfiability Modulo Theories Competition.) SPEAR has 26 parameters including ten categorical, four integer, and twelve continuous parameters, and their default values were hand-tuned by its developer. The categorical parameters mainly control heuristics for variable and value selection, clause sorting, resolution ordering, and enable or disable optimizations, such as the pure literal rule. The continuous and integer parameters mainly deal with activity, decay, and elimination of variables and clauses, as well as with the interval of randomized restarts and percentage of random choices. We discretized the integer and continuous parameters by choosing lower and upper bounds at reasonable values and allowing between three and eight discrete values spread relatively uniformly across the resulting interval, including the default. As the starting configuration for ParamILS, we used the default. The number of discrete values was chosen according to our intuition about the importance of the parameter. After this discretization, there were 3.7×10^{18} possible parameter configurations. However, exploiting the fact that nine of the parameters are conditional (i.e., only relevant when other parameters take certain values) we reduced the total to 8.34×10^{17} configurations.

CPLEX The third algorithm is the commercial optimization tool CPLEX 10.1.1, a massively parameterized algorithm for solving mixed integer programming (MIP) problems. Out of its 159 user-specifiable parameters, we identified 81 parameters that affect CPLEX's search trajectory. A large number of parameters deal with MIP strategy heuristics (such as variable and branching heuristics, probing, dive type, and subalgorithms) and amount and type of preprocessing to be performed. There are also nine parameters each governing how frequently a different type of cut should be used (there are four magnitude values and the value "choose automatically"; note that this last value prevents the parameters from being ordinal). A considerable number of other parameters deal with simplex and barrier optimization and various other algorithm components. Many of the parameters have automatic settings, and in the case of numerical parameters we chose these automatic settings instead of hypothesizing values that might work well. We also identified a number of numerical parameters that dealt primarily with numerical issues, and fixed those to their default values. For other numerical parameters, we chose up to five possible values that seemed sensible, including the default. For the many categorical parameters with an automatic option, we included the automatic option as a choice for the parameter, but also included all the manual options. Finally, we ended up with 63 configurable parameters, leading to 1.78×10^{38} possible configurations. Exploiting the fact that seven of the CPLEX parameters were only relevant conditional on other parameters taking certain values, we reduced this to 1.38×10^{37} distinct configurations. As the starting configuration for our configuration procedures, we used the default settings, which were reportedly obtained by careful manual configuration on a broad range of MIP instances.

3.2.2 BENCHMARK INSTANCES

In our INDEPTH configuration scenarios, we applied our target algorithms to three sets of benchmark instances: SAT-encoded quasi-group completion problems, SAT-encoded graph-colouring problems based on small world graphs, and MIP-encoded winner determination problems for combinatorial auctions. (In our BROAD configuration scenarios, we employ a number of additional benchmark instances. These are described in detail in Section 8.1.) In our INDEPTH scenarios, each set consisted of 2000 instances, divided randomly into training and test sets of 1,000 instances each.

QCP Our first benchmark set contained 23 000 instances of the quasi-group completion problem (QCP), which has been widely studied by AI researchers. We generated these QCP instances around the solubility phase transition, using the parameters given by Gomes and Selman (1997). Specifically, the order n was drawn uniformly from the interval $[26, 43]$, and the number of holes H (open entries in the Latin square) was drawn uniformly from $[1.75, 2.3] \times n^{1.55}$; finally, these QCP instances were converted into SAT CNF format. For use with the complete solver, SPEAR, we sampled 2000 of these SAT instances uniformly at random; these had on average 1497 variables (standard deviation: 1094) and 13 331 clauses (standard deviation: 12 473), and 1182 of them were satisfiable. For use with the incomplete solver, SAPS, we randomly sampled 2000 instances from the subset of satisfiable instances (determined using a complete algorithm); their number of variables and clauses were very similar to those used with SPEAR.

SW-GCP Our second benchmark set contained 20 000 instances of the graph colouring problem (GCP) based on the small world (SW) graphs of Gent et al. (1999). Of these, we sampled 2000 instances uniformly at random for use with SPEAR; these had on average 1813 variables (standard deviation: 703) and 13 902 clauses (standard deviation: 5393), and 1109 of them were satisfiable. For use with SAPS, we randomly sampled 2000 satisfiable instances (again, determined using a

complete SAT algorithm), whose number of variables and clauses were very similar to those used with SPEAR.

Regions100 For our third benchmark set we generated 2,000 combinatorial winner determination instances, encoded as mixed-integer linear programs (MILPs). We used the generator provided with the Combinatorial Auction Test Suite (Leyton-Brown et al., 2000), based on the `regions` option with the goods parameter set to 100 and the bids parameter set to 500. The resulting MILP instances contained 501 variables and 193 inequalities on average, with a standard deviation of 1.7 variables and 2.5 inequalities.

3.3 Experimental Setup

All of our experiments were carried out on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running OpenSuSE Linux 10.1. We measured runtimes as CPU time on these reference machines. In our five INDEPTH configuration scenarios, we set fairly aggressive cutoff times of five seconds per run of the target algorithm and allowed each configuration procedure to execute the target algorithm for an aggregate runtime of five CPU hours. All our configuration procedures are implemented as Ruby scripts, and we do not include the runtime of these scripts in the configuration time. In "easy" configuration scenarios, where algorithm runs finish in milliseconds, the overhead of our scripts can be substantial, and one configuration run indeed took 24 hours in order to execute five hours worth of target algorithm runtime. In contrast, for the harder BROAD scenarios we found virtually no overhead. For the self-configuration experiments in Section 9 overhead on "easy" configuration scenarios was problematic. As a consequence, for the experiments in that Section we optimized our scripts, producing a new version that was significantly faster. (Both versions of the scripts are available online at `http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/`.)

For the nine BROAD configuration scenarios in Section 8, we terminated unsuccessful runs after 300 CPU seconds and allowed each configuration procedure to execute the target algorithm for an aggregate runtime of two CPU days. Overall, we spent about 3.4 CPU years gathering the data analyzed in this paper: about 170 CPU days for the results in Section 4, 210 CPU days for Sections 5–7, 360 CPU days for Section 8 and 500 CPU days for Section 9.

4. Study of Problem Characteristics Based on Random Sampling

In this section, we study our five INDEPTH configuration scenarios independently of our ParamILS framework. Specifically, we describe a simple uniform random search procedure, and use it to explore both the variations in performance across parameter configurations and the variations in (algorithm-specific) hardness across the instances in the training set.

4.1 The RandomSearch procedure

A very simple way to build an automatic configurator is to iteratively determine a configuration by sampling uniformly at random, to evaluate it on N instances from the training set, and to keep track of the best configuration thus encountered. We call this procedure *RandomSearch* and give pseudo-code for it in Algorithm 1.

In the remainder of this section, we use the RandomSearch procedure to gain initial understanding of our configuration scenarios and to understand the sensitivity of our target algorithms to their parameters.

Algorithm 1: RandomSearch(N, θ_0)

Outline of random search in parameter configuration space; θ_{inc} denotes the incumbent parameter configuration, *better*$_N$ compares two configurations based on the first N instances from the training set.

Input : Number of runs to use for evaluating parameter configurations, N; initial configuration $\theta_0 \in \Theta$.

Output : Best parameter configuration θ^* found.

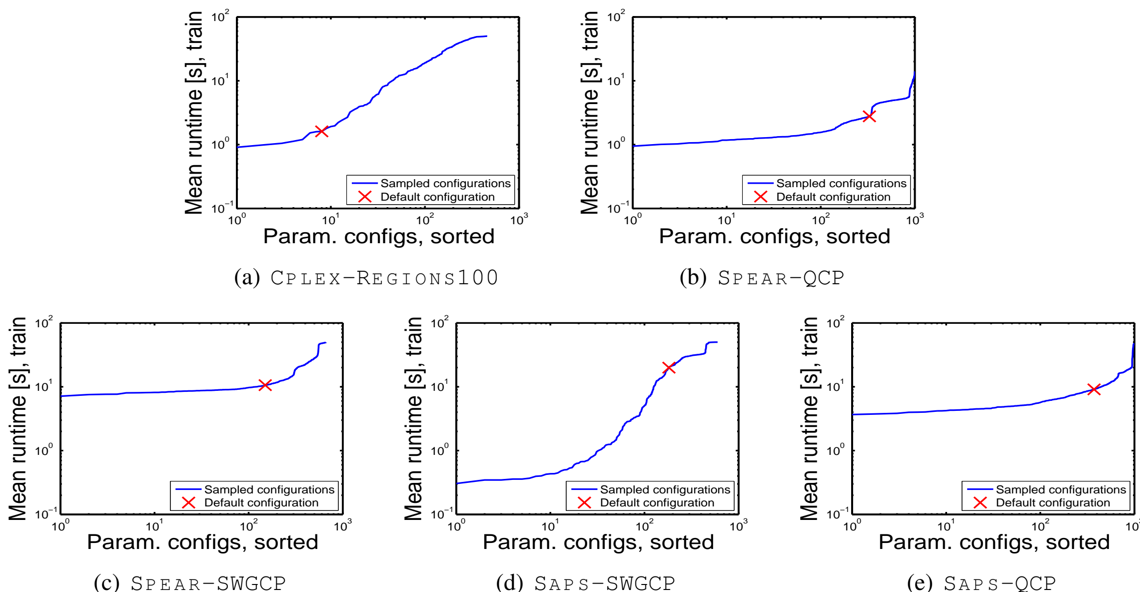
```
1 θ_inc ← θ_0;
2 while not TerminationCriterion() do
3     θ ← random θ ∈ Θ;
4     if better_N(θ, θ_inc) then
5         θ_inc ← θ;
6 return θ_inc
```

4.2 Performance Variation across Parameter Configurations and Instances

Figure 2 shows the performance variation across all configurations considered by RandomSearch for each configuration scenario, as well as the respective default parameter configurations. Note that in all configuration scenarios a significant number of parameter configurations beat the default, in some cases by a large margin. For example, in the case of SAPS-SWGCP, the default configuration yields an average runtime of around 20 seconds, while the sampled configuration with the best performance takes an average of 0.1 seconds per instance. Recall that we count the runtime for unsuccessful runs as ten times the cutoff time; thus, a mean runtime of 20 seconds for the default implies that a fair percentage (between 30% and 40%) of the runs were unsuccessful in the first five seconds and therefore counted at $5 \times 10 = 50$ seconds each when computing mean runtime. Also note that for the highly optimized CPLEX algorithm the performance of the default parameter configuration is closer to the best of the sampled configurations, but that there are still a number of better-performing randomly sampled parameter configurations.

Figure 3 sheds light on how these differences in mean runtime come about. For each configuration scenario and for six parameter configurations each, the figure gives a cumulative distribution of benchmark instances solved as a function of time. The six configurations shown are the default, the best sampled configuration, the worst sampled configuration, and the configurations representing the $q_{0.25}$, $q_{0.50}$, and $q_{0.75}$ quantiles. All of these evaluations of configuration quality (best, worst, etc.) are calculated with respect to the training benchmark set of 1,000 instances. (In cases where the worst parameter configuration does not solve *any* of the 1,000 instances, we do not show it in the figure.)

We notice substantial differences across the five INDEPTH configuration scenarios, especially in the variation of (algorithm-specific) instance hardness. The QCP distribution contains a substantial portion of trivially solvable instances, both for the local search algorithm SAPS and the tree search algorithm SPEAR. We also observe that the SWGCP distribution seems quite easy for SAPS with an appropriate parameter configuration, and that for all configuration scenarios the variability in instance hardness depends substantially on the parameter configuration used. Instance distribution Regions100 shows the lowest variability, in that the difference in hardness between the easiest and the hardest instance is “only” about an order of magnitude for the best parameter configuration.



(a) CPLEX-REGIONS100 (b) SPEAR-QCP

(c) SPEAR-SWGCP (d) SAPS-SWGCP (e) SAPS-QCP

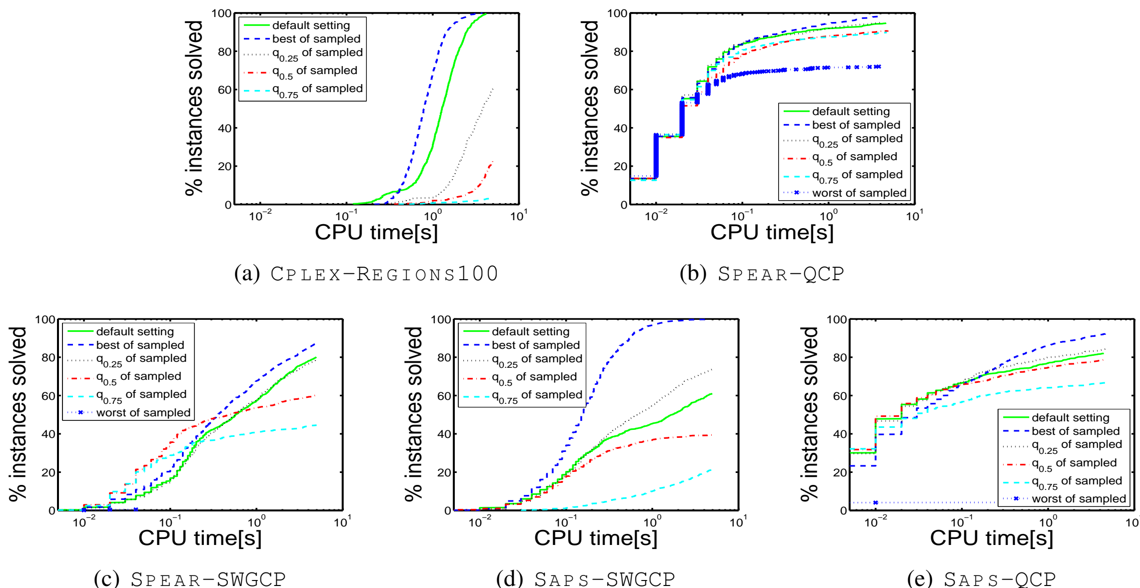
Figure 2: Variation of performance (mean runtime over 1,000 training instances) across parameter configurations in different configuration scenarios. Performance of the default parameter configuration and a number of randomly sampled configurations; the number of sampled configurations for each scenario is the number of configurations that could be evaluated within 500 CPU hours, up to a maximum of 1,000; note the log-log scale.

In using the mean runtime as our configuration objective, we are implicitly looking for parameter configurations with robust performance across instances. This is not the case if we, for example, choose to optimize the median runtime or some other runtime quantile: for example, in Figure 3(c), we would prefer configuration $q_{0.5}$ over the default configuration when minimizing *median* runtime (this is even clearer for the 40% quantile), but prefer the default configuration when minimizing *mean* runtime.

4.3 Overconfidence of Training Performance

So far, we have seen that random search in the space of feasible parameter configurations can identify configurations that are much better than the default configuration when given enough time to evaluate a large number of parameter configurations using a large number of $N = 1,000$ runs to evaluate each configuration. Next, we studied RandomSearch as an actual configuration procedure, measuring the performance of its incumbent throughout its trajectory. Training performance is measured as mean runtime (in CPU seconds) across the chosen training instances (different in each repetition of the configurator), while test performance is measured as mean runtime across the 1,000 fixed test instances.

Figure 4 shows the training performance of RandomSearch(100) compared to the performance of the default parameter configuration. In particular, we employed 100 repetitions of RandomSearch(100) that all used the same randomly sampled configurations as above, but in a different



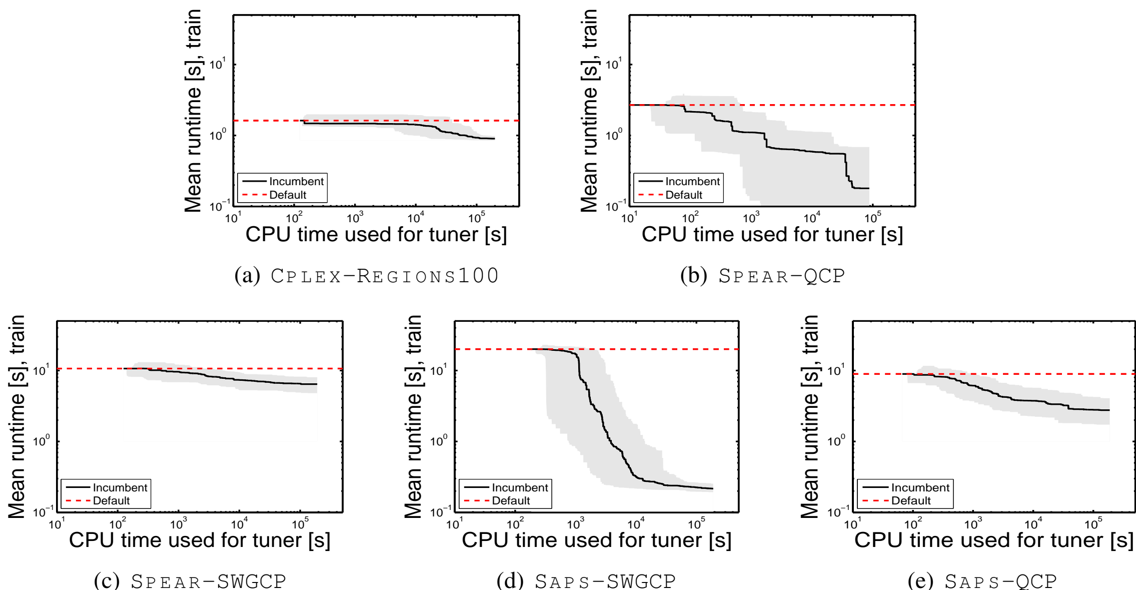
(a) CPLEX-REGIONS100 (b) SPEAR-QCP

(c) SPEAR-SWGCP (d) SAPS-SWGCP (e) SAPS-QCP

Figure 3: Hardness variation across instances in different configuration scenarios. For each scenario, the plot shows the percentage of benchmark instances solved by a number of parameter configurations within a given time. We plot the curves for the default parameter configuration, and five of the randomly sampled parameter configurations, in particular the best, the worst, and the $q_{0.25}$, $q_{0.50}$, and $q_{0.75}$ quantiles, all with respect to the training benchmark set of 1,000 instances.

order; each repetition also used a different set of $N = 100$ randomly sampled training instances (out of the full training set of 1,000 instances). For each time point, we then collected the mean runtime on the training instances for the incumbent of each of the 100 repetitions, and plotted mean $\pm$ standard deviation. Note that in each of our configuration scenarios, this simple method found parameter configurations that performed better than the default on the training benchmark set, sometimes substantially so.

To be practical, however, a configuration procedure must yield parameter configurations that not only do well on instances used for training, but also on previously unseen test instances. In some cases, a configurator may have identified parameter settings that are adapted specifically to the instances in the training set and thus not achieve good performance on previously unseen test instances. We call this phenomenon *overconfidence* of the training performance. This effect is very well known in machine learning where training set results are well-known to be unreliable (and usually optimistic) estimators of test set performance (Hastie et al., 2001). In some scenarios, test set performance actually *degrades* when the number of considered parameter settings (in machine learning the *hypothesis space*) is increased beyond a certain point; this effect is called *over-fitting* in machine learning and *over-tuning* in optimization (Birattari et al., 2002; Birattari, 2005; Hutter et al., 2007b).



(a) CPLEX-REGIONS100 (b) SPEAR-QCP

(c) SPEAR-SWGCP (d) SAPS-SWGCP (e) SAPS-QCP

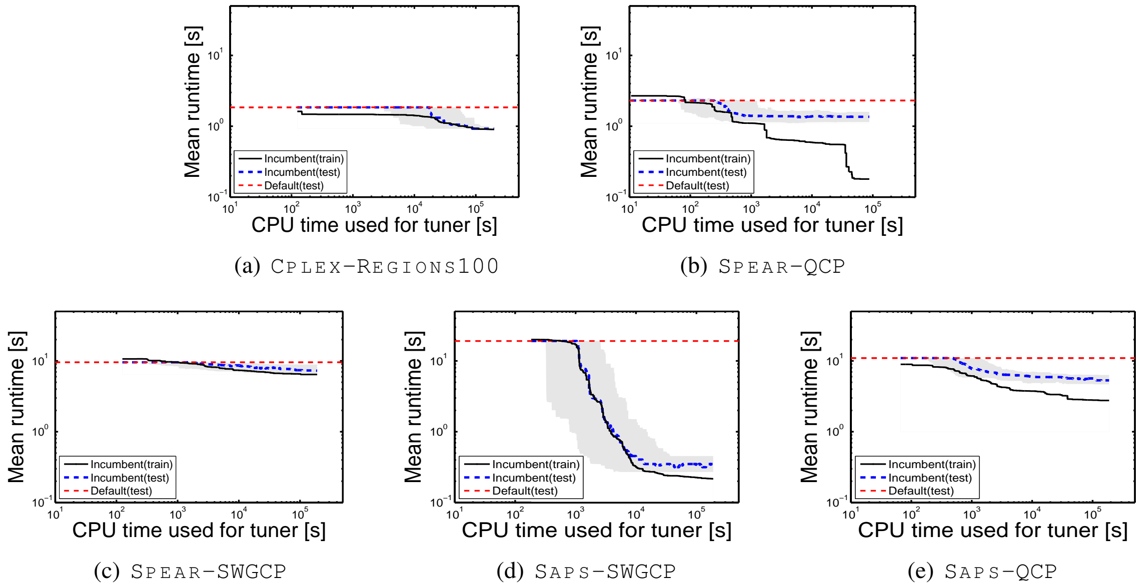
Figure 4: Training performance (mean runtime over $N = 100$ training instances) of RandomSearch(100) for our INDEPTH scenarios. We performed 25 repetitions of RandomSearch and plot mean $\pm$ stddev (denoted by the grey region) of training performance over these repetitions; for reference, we plot training performance of the default parameter configuration.

Figure 5 shows test performance for the exact same parameter configurations selected in Figure 4. For reference, the plots also indicate the mean training performance. For the QCP configuration scenarios there is clear evidence for overconfidence (divergence between training and test performance), while this effect is much smaller for the other scenarios. We attribute this to the higher variability in instance hardness for the QCP distributions as indicated in Figure 3.

4.4 Trading off Runtime, Number of Training Instances, and Captime

In the previous section, we saw that in each configuration scenario, the simple procedure RandomSearch(100) can identify configurations that already perform better than the respective default configurations. This speedup was larger on the instances the configurator used for training but also generalized to test instances unknown to the configurator in all scenarios. However, in these initial experiments the configurator was allowed to run for a long time of 50 CPU hours. This is despite the fact that the configuration scenarios have low captimes of only $\kappa = 5s$ for each single run.

The time required for the evaluation of a parameter configuration is upper-bounded by $N \times \kappa$, and we can reduce this bound by decreasing N, κ, or both. However, these speedups come at the price of potentially worse generalizations to unseen test instances and higher runtimes during testing, respectively. In this section, we explore this trade-off by varying both N and κ. Throughout this section, we use exactly the same training and test data, as well as the same sequence of parameter configurations as used in the previous section to produce Figure 5. In order to evaluate the



(a) CPLEX-REGIONS100 (b) SPEAR-QCP

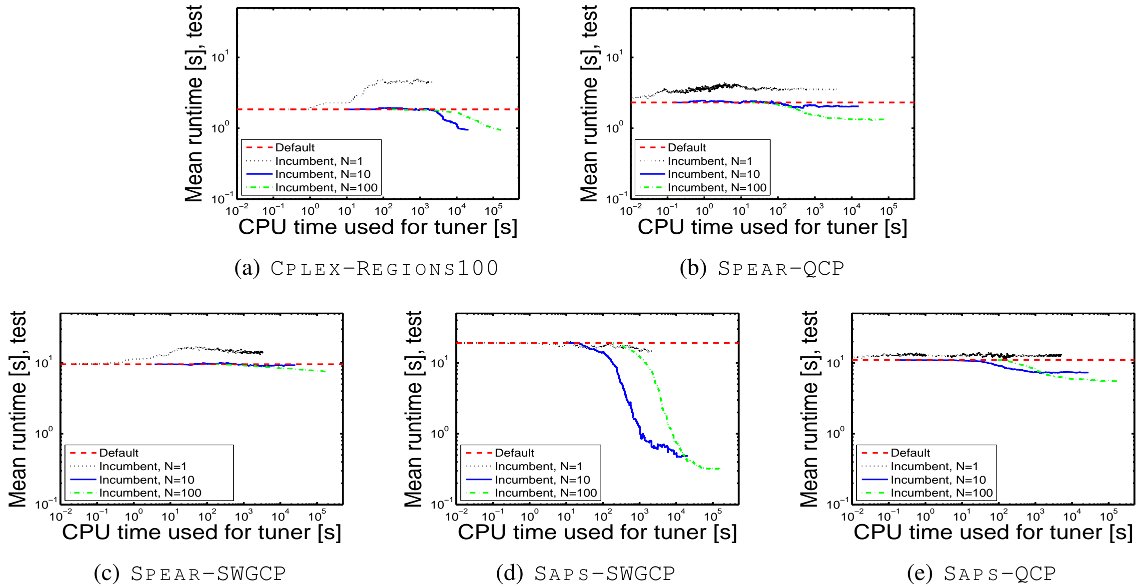
(c) SPEAR-SWGCP (d) SAPS-SWGCP (e) SAPS-QCP

Figure 5: Test performance (mean runtime over 1,000 test instances) of RandomSearch(100) for our INDEPTH scenarios, compared to training performance (mean runtime over $N = 100$ training instances). We performed 25 repetitions of the configurator and plot mean test performance $\pm$ stddev of test performance, as well as mean training performance; for reference, we plot test performance of the default parameter configuration.

effect of changing the number of instances or the captime, we simply interpreted this data differently, pretending that only a smaller number of instances N was available, or that the captime κ was lower.

In Figure 6, we plot the test performance that resulted from using training benchmark sets of size $N = 1$,10, and 100. The expected amount of time needed for the evaluation of a single parameter configuration is proportional to the number of training instances, N. Thus, it is not surprising that the optimization runs only using a single training instance were much faster than those using more instances. We reiterate that all approaches were evaluated on the same number of configurations; thus, the curves for $N = 1$ ends at a time roughly 10 times earlier than $N = 10$, which in turn ends roughly 10 times earlier than the curve for $N = 100$.

We observe that generalization performance from a single instance was very poor, typically leading to test performance worse than that of the default. A training benchmark set with $N = 10$ instances yielded fairly fast optimization runs that quickly achieved good generalization performance in some scenarios (see, e.g., scenario CPLEX-REGIONS100 in Figure 6(a)). Using $N = 100$ training instances yielded substantially better test performance than $N = 10$ for all scenarios except CPLEX-REGIONS100, but took much longer to do so. Thus, we conclude that the optimal fixed number of training instances to use depends on how much time is available for configuration. For reference, in our experiments described later on in this article, we allocated 5 CPU hours for configuring these INDEPTH scenarios. This coincides with the time at which the plots for $N = 10$ end.



(a) CPLEX-REGIONS100 (b) SPEAR-QCP

(c) SPEAR-SWGCP (d) SAPS-SWGCP (e) SAPS-QCP

Figure 6: Test performance (mean runtime over 1,000 test instances) of RandomSearch(N) for our INDEPTH scenarios, for $N = 1$, 10, and 100. We performed 25 repetitions of RandomSearch(N) for each N and plot mean test performance; for reference, we plot test performance of the default parameter configuration.

At this time, $N = 10$ yielded the best result for scenario CPLEX-REGIONS100 and roughly tied with $N = 100$ for scenario SAPS-SWGCP; for the other three scenarios $N = 100$ performed best.

Next, we studied the dependence of a configurator's test performance on the used captime κ (the maximum time allowed for each run of the target algorithm); to the best of our knowledge, this is the first study of its kind. Our experimental results are reported in Figure 7. We allowed the full cutoff time (5 seconds), a tenth of it (0.5 seconds), and a hundredth of it (0.05 seconds). As with using very few training instances, using a very low κ of 0.05 seconds led to a very fast search with poor generalization performance (this is most apparent for scenario CPLEX-REGIONS100 in Figure 7(a)). An intermediate value of $\kappa = 0.5$ seconds still yielded a fast search, sometimes leading to good generalizations quickly (see, e.g., scenario SAPS-SWGCP in Figure 7(d)). Finally, with the highest value of κ, 5 seconds, the best results were achieved, but at the price of longer runtime. When evaluated at a configuration time of 5 hours (which coincides with the plot for $\kappa = 0.5$ ending), $\kappa = 0.5s$ yielded slightly better results for scenario SAPS-SWGCP and tied with $\kappa = 5s$ for scenario SPEAR-QCP; $\kappa = 5s$ was the clear winner for scenario CPLEX-REGIONS100 and best by a small margin for SPEAR-SWGCP and SAPS-QCP. Overall, we observe a phenomenon similar to overconfidence: when using very low cutoff times, parameter configurations are chosen that perform well for short cutoff times (i.e., solve a larger percentage of instances than other parameter configurations). However, these configurations might not solve many more instances when allowed a larger cutoff time; instead, it might be better to choose parameter configurations that perform poorly initially but excel when given a large enough cutoff time.

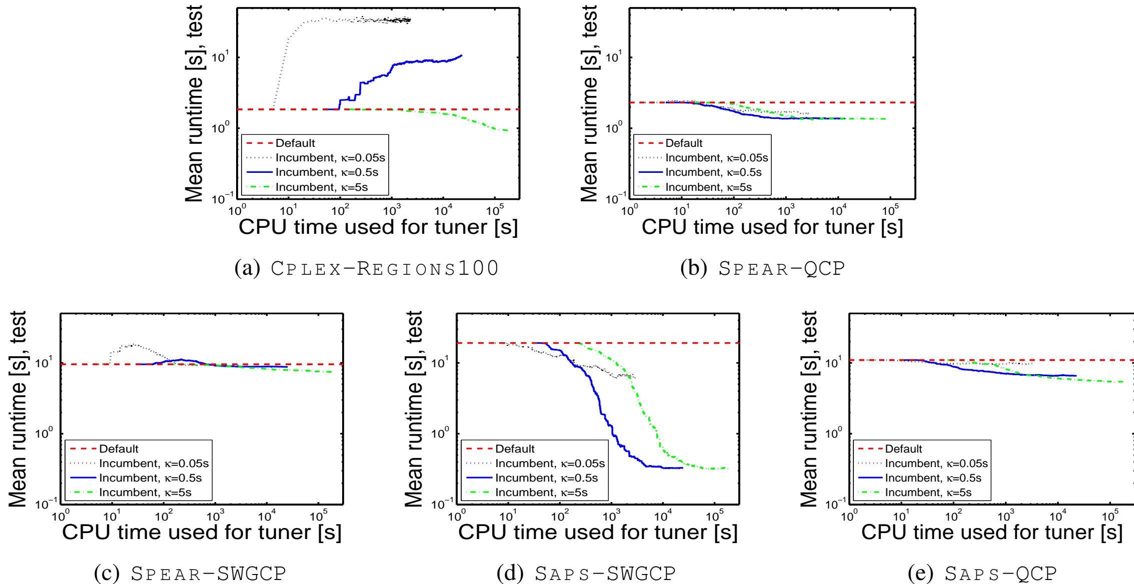
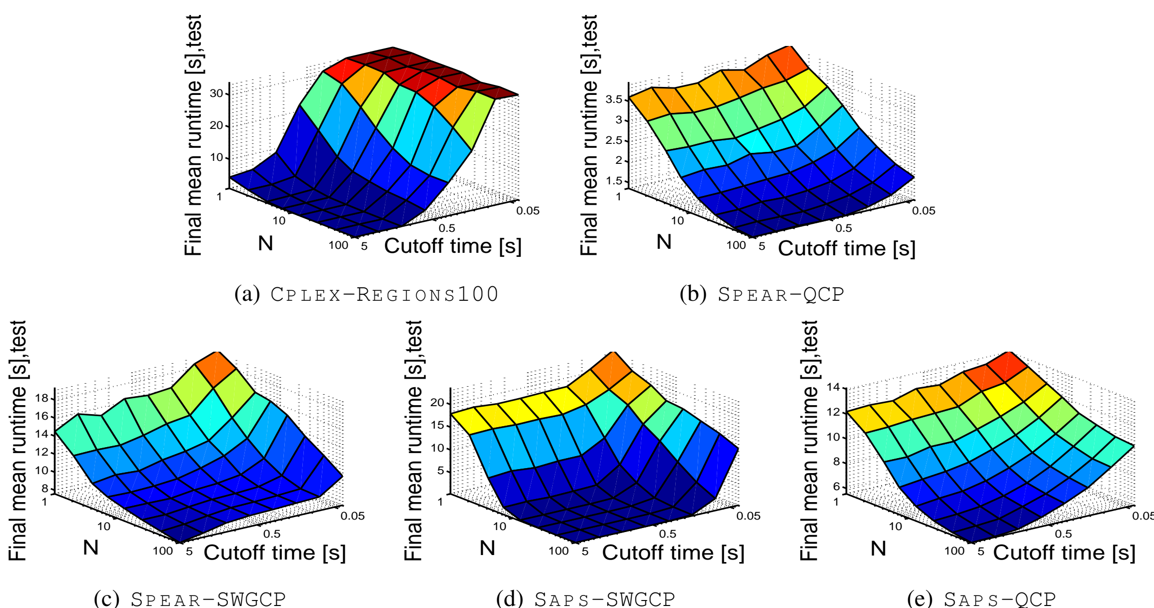


Figure 7: Test performance (mean runtime over 1,000 test instances) of RandomSearch(100) for our INDEPTH scenarios, using different cutoff times ($\kappa = 0.05s$, $\kappa = 0.5s$, and $\kappa = 5s$). We performed 25 repetitions of RandomSearch(100) for each κ and plot mean test performance; for reference, we plot test performance of the default parameter configuration.

As we have seen, the result of the algorithm configuration procedure in terms of target algorithm performance depends on the number of training instances, N, and the cutoff time per instance, κ. In a final experiment, we explored this trade-off by varying both N and κ at the same time, and measuring test performance of the final incumbent for each combination of N and κ.

Figure 8 shows the results of this experiment, illustrating the important tradeoff between cutoff time and number of training instances. Recall that we use the same runtime data for each combination. Since the runtime for a single evaluation is upper-bounded by $N \times \kappa$, the total runtime differs between the combinations. For example, while the runtime for $N = 100$ and $\kappa = 5s$ was 50 CPU hours, it was roughly 0.5 CPU hours for $N = 10$ and $\kappa = 0.5s$. Thus, unsurprisingly, the best results were achieved with the maximal number of training instances ($N = 100$) and the maximal cutoff time ($\kappa = 5$ seconds) and degraded monotonically with lower N and κ.

From the Figure, we observe major differences between the configuration scenarios in terms of the relative effects of N and κ. For example, for scenario SPEAR-QCP (see Figure 8(b)) the number of training instances was much more important than the cutoff time (decreasing cutoff time did not worsen performance as much). On the other hand, for CPLEX-REGIONS100 (see Figure 8(a)) this relationship was reversed, with the cutoff time becoming a crucial component and the number of training instances being fairly inconsequential. The importance of a large number of training instances in configuration scenario SPEAR-QCP can be explained by the high variance in instance hardness as shown in Figure 3(b), which also shows that, when using a large number of



(a) CPLEX-REGIONS100 (b) SPEAR-QCP

(c) SPEAR-SWGCP (d) SAPS-SWGCP (e) SAPS-QCP

Figure 8: Test performance (mean runtime over 1,000 test instances) of the best parameter configuration found with RandomSearch, as depending on the number of training instances, N, and the cutoff time, κ, used. Note that each combination of N and κ was allowed to evaluate the same number of configurations, and that the time required for each combination was thus roughly proportionally to $N \times \kappa$. Also note the non-standard orientation of the axes with large values of N and cutoff time towards the front of the 3D plot (chosen to avoid clutter).

instances, the best parameter configuration with a cutoff time of five seconds was also the best configuration when using much lower cutoff times, such as 0.1 seconds. This is not the case for scenario CPLEX-REGIONS100: as shown in Figure 3(a), not a single parameter configuration even solves a single instance with a cutoff time as low as 0.1 seconds. Thus, in this configuration scenario comparisons based on such low cutoff times reverted to blind guessing.

5. ParamILS: Iterated Local Search in Parameter Configuration Space

In this section, we address the first and most important of the previously mentioned dimensions of automated algorithm configuration, the search strategy. For now, we fix the other two dimensions, using an unvarying benchmark set of instances and fixed cutoff times for the evaluation of each parameter configuration. Thus, the stochastic optimization problem of algorithm configuration reduces to a simple optimization problem, namely to find the parameter configuration that yields the lowest mean runtime on the given benchmark set.

In Section 4.1, we introduced the configuration procedure RandomSearch. This method was convenient due to its simplicity and as a way of investigating issues that are important for *any* approach for algorithm configuration. However, as one may expect, there are more effective methods

for searching parameter configuration space. In this section, we describe one such method, an iterated local search framework called ParamILS, and compare its effectiveness in searching parameter configuration space to that of RandomSearch. We also demonstrate that local optima in parameter configuration space exist, and that ParamILS deals effectively with them.

5.1 The ParamILS framework

Consider the following manual parameter optimization process:

1. begin with some initial parameter configuration;

2. experiment with modifications to single parameter values, accepting new configurations whenever they result in improved performance;

3. repeat step 2 until no single-parameter change yields an improvement.

This widely used procedure corresponds to a manually-executed local search in parameter configuration space. Specifically, it corresponds to an iterative first improvement procedure with a search space consisting of all possible configurations, an objective function that quantifies the performance achieved by the target algorithm with a given configuration and a neighbourhood relation under which each step consists of the modification of a single parameter (a so-called "one-exchange" neighbourhood).

Viewing this manual procedure as a local search algorithm is advantageous because it suggests the automation of the procedure, and also the procedure's improvement by drawing on ideas from the stochastic local search community. For example, note that the procedure stops as soon as it reaches a local optimum (a parameter configuration that cannot be improved by modifying a single parameter value). A more sophisticated approach is to employ iterated local search (ILS) (Lourenço et al., 2002) to search for performance-optimizing parameter configurations. ILS is a stochastic local search method that builds a chain of local optima by iterating through a main loop consisting of (1) a solution perturbation to escape from local optima, (2) a subsidiary local search procedure and (3) an acceptance criterion that is used to decide whether to keep or reject a newly obtained candidate solution.

ParamILS (given in pseudocode as Algorithm 2) is an ILS method that searches parameter configuration space. It uses a combination of default and random settings for initialization, employs iterative first improvement as a subsidiary local search procedure, uses a fixed number (s) of random moves for perturbation, and always accepts better or equally-good parameter configurations, but re-initializes the search at random with probability $p_{restart}$.[2] Furthermore, it is based on a one-exchange neighbourhood, that is, we always consider changing only one parameter at a time. ParamILS deals with conditional parameters by excluding all configurations from the neighbourhood of a configuration θ that differ only in a conditional parameter that is not relevant in θ.

5.2 The BasicILS Algorithm

In order to turn ParamILS into an executable configuration procedure, it is necessary to augment the framework given in Algorithm 2 with a function *better* that determines which of two parameter settings should be preferred. We will ultimately propose several different ways of doing this.

2. Our original choices of $\langle r, s, p_{restart} \rangle = \langle 10, 3, 0.01 \rangle$ from (Hutter et al., 2007b) were somewhat arbitrary, though we expected performance to be quite robust with respect to these settings. We revisit this issue in Section 9.

Algorithm 2: ParamILS($\theta_0, r, p_{restart}, s$)

Outline of iterated local search in parameter configuration space; the specific variants of ParamILS we study, *BasicILS(N)* and *FocusedILS*, differ in the procedure *better* (which compares $\theta, \theta' \in \Theta$).

```
   Input  : Initial configuration θ0 ∈ Θ, algorithm parameters r, p_restart, and s.
   Output : Best parameter configuration θ found.
1  for i = 1, ..., r do
2      θ ← random θ ∈ Θ;
3      if better(θ, θ0) then θ0 ← θ;
4  θ_ils ← IterativeFirstImprovement (θ0, N);
5  while not TerminationCriterion() do
6      θ ← θ_ils;
       // ===== Perturbation
7      for i = 1, ..., s do θ ← random θ' ∈ N(θ);
       // ===== Basic local search
8      θ ← IterativeFirstImprovement (θ, N);
       // ===== AcceptanceCriterion
9      if better(θ, θ_ils) then θ_ils ← θ;
10     with probability p_restart do θ_ils ← random θ ∈ Θ;
11 return overall best θ found;
12 Procedure IterativeFirstImprovement (θ, N)
13 repeat
14     θ' ← θ;
15     foreach θ'' ∈ N(θ') in randomized order do
16         if better(θ'', θ') then θ ← θ''; break;
17 until θ' = θ;
18 return θ;
```

Procedure 3: better$_N(\theta_1, \theta_2)$

Procedure used in RandomSearch(N) and BasicILS(N) to compare two parameter configurations. Procedure objective(θ, N) returns the user-defined objective achieved by $\mathcal{A}_\theta$ on the first N instances and keeps track of the incumbent solution; it is detailed in Procedure 5 on page 31.

```
  Input  : Parameter configuration θ1, parameter configuration θ2
  Output : True if θ1 does better than or equal to θ2 on the first N instances; false otherwise
1 ĉ_N(θ2) ← objective(θ2, N)
2 ĉ_N(θ1) ← objective(θ1, N)
3 return ĉ_N(θ1) ≤ ĉ_N(θ2)
```

Here, we describe the simplest approach, which we call *BasicILS*. Specifically, we use the term BasicILS(N) to refer to a ParamILS algorithm in which the function *better*(θ_1, θ_2) is implemented as shown in Procedure 3: simply compare estimates $\hat{c}_N$ of the cost statistics $c(\theta_1)$ and $c(\theta_2)$ that are based on N runs each. (This is the same procedure that is used to compare parameter configurations in RandomSearch(N).)

BasicILS(N) is a simple and intuitive approach since it evaluates every parameter configuration by running it on the same N training benchmark instances using the same seeds. Like many other

related approaches (see, e.g., (Minton, 1996; Coy et al., 2001; Adenso-Diaz and Laguna, 2006)) it circumvents the stochastic part of the optimisation problem by using an estimation based on a fixed training set of N instances, effectively treating the problem as a standard optimisation problem. When benchmark instances are very heterogeneous or when the user can identify a rather small "representative" subset of instances, this approach can find good parameter configurations with low computational effort. As we will see, BasicILS can achieve impressive performance in some configuration scenarios.

5.3 Experimentally Evaluating BasicILS

In this section, we evaluate the effectiveness of BasicILS(N) against two of its components, RandomSearch(N) (used in BasicILS(N) for initialization) and a simple local search (the same type of iterative first improvement search used in BasicILS(N); we dub it SimpleLS(N)). If there is sufficient structure in the search space, BasicILS should outperform RandomSearch; furthermore, if there are local minima, BasicILS should perform better than basic local search. Our experiments showed that BasicILS did indeed offer the best performance.

We set the cutoff time to five seconds per run and the number of training instances to 100. For all configuration procedures, we performed 25 independent repetitions, each of them with a different training set of 100 instances (constructed as described in Section 3.1.2).

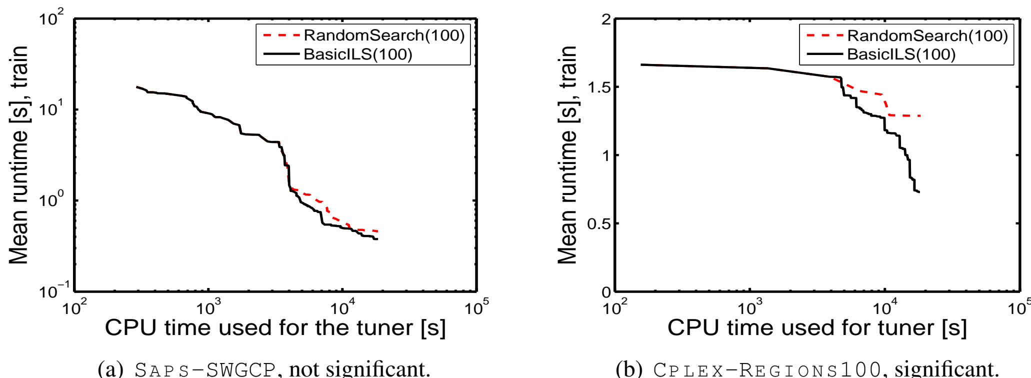
At this point in the paper, we are solely interested in comparing how effectively the approaches search the space of parameter configurations (and not how the found parameter configurations generalize to unseen test instances). Thus, we compare training performance achieved.

First, we present our comparison of BasicILS against RandomSearch. In Figure 9, we plotted the mean solution quality achieved by the two approaches at a given time, for the two configuration scenarios with the least and the most pronounced differences between the two configuration procedures: SAPS-SWGCP and CPLEX-REGIONS100. BasicILS started with $r = 10$ random samples (using the same seed as RandomSearch), which meant that performance for the first part of the trajectory was always identical. After these random samples, BasicILS performed better, quite clearly so for CPLEX-REGIONS100. Table 3 quantifies the performance of both approaches on our INDEPTH configuration scenarios. BasicILS always performed better, and in three of the five scenarios the difference was statistically significant as judged by a paired Max-Wilcoxon test (see Section 3.1.3). Table 9 also lists the performance of the default parameter configuration for each of the configuration scenarios. We note that both BasicILS and RandomSearch consistently made substantial (and statistically significant) improvements over these default configurations.

Next, we compared BasicILS against its second component, SimpleLS. This basic local search is identical to BasicILS, but stops in the first local minimum encountered. We used it in order to study whether local minima are present that pose a problem for simple first improvement search. Table 4 shows that in the three configuration scenarios where BasicILS had time to perform multiple iterations, its training set performance was statistically significantly better than that of SimpleLS. Thus, we conclude that the search space contains structure that can be exploited with a local search algorithm as well as local minima that can limit the performance of iterative improvement search.

6. FocusedILS: Adaptively Selecting the Number of Training Instances

In this section, we go beyond BasicILS by addressing the second dimension of automated algorithm configuration, choosing the number of training instances. In Section 4.3, we described the phe-



(a) SAPS-SWGCP, not significant.

(b) CPLEX-REGIONS100, significant.

Figure 9: Comparison of training performance (mean runtime over $N = 100$ training instances) of BasicILS(100) and RandomSearch(100) for the configuration scenarios in which the difference between the two approaches was least significant and most significant. We performed 25 repetitions of each configurator and at each time step plot the median training performance of these repetitions (plots are similar for *means* over 25 repetitions, but more variable and thus cluttered). Note the logarithmic scale on the x-axis, and the difference in y-axis scales: we chose a log-scale for SAPS-SWGCP due the large performance variation seen for this scenario, and a linear scale for CPLEX-REGIONS100, where even poor configurations performed quite well. We will use these two configuration scenarios for visualization purposes throughout, always using the same y-axes as in this plot.

Scenario	Default	RandomSearch(100)	BasicILS(100)	p-value
SAPS-SWGCP	19.93	0.46 ± 0.34	$\mathbf{0.38 \pm 0.19}$	0.94
SPEAR-SWGCP	10.61	7.02 ± 1.11	$\mathbf{6.78 \pm 1.73}$	0.18
SAPS-QCP	9.11	3.73 ± 1.53	$\mathbf{2.86 \pm 1.25}$	$\mathbf{1.8 \times 10^{-5}}$
SPEAR-QCP	2.77	0.58 ± 0.59	$\mathbf{0.36 \pm 0.39}$	$\mathbf{0.007}$
CPLEX-REGIONS100	1.61	1.45 ± 0.35	$\mathbf{0.72 \pm 0.45}$	$\mathbf{1.2 \times 10^{-5}}$

Table 3: Comparison of final training performance (mean runtime over $N = 100$ training instances) for RandomSearch(100) and BasicILS(100). We performed 25 repetitions of each configurator, using identical ⟨instance, seed⟩ lists for both configurators (but different ⟨instance, seed⟩ lists for each of the 25 repetitions). Note that both approaches yielded substantially better results than the default configuration, and that BasicILS performed statistically significantly better than RandomSearch in three of the five INDEPTH configuration scenarios as judged by a paired Max-Wilcoxon test (see Section 3.1.3).

Scenario	SimpleLS(100)	BasicILS(100)		p-value
	Performance	Performance	Avg. # iterations	
SAPS-SWGCP	0.5 ± 0.39	$\mathbf{0.38 \pm 0.19}$	2.6	$\mathbf{9.8 \times 10^{-4}}$
SAPS-QCP	3.17 ± 1.42	$\mathbf{2.86 \pm 1.25}$	4.84	$\mathbf{4.9 \times 10^{-4}}$
SPEAR-QCP	0.4 ± 0.39	$\mathbf{0.36 \pm 0.39}$	1.64	**0.008**

Table 4: Comparison of final training performance (mean runtime over $N = 100$ training instances) for SimpleLS(100) and BasicILS(100). We performed 25 repetitions of each configurator, using identical $\langle$instance, seed$\rangle$ lists for both configurators (but different $\langle$instance, seed$\rangle$ lists for each of the 25 repetitions). In configuration scenarios SPEAR-SWGCP and CPLEX-REGIONS100, BasicILS did not complete its first iteration in any of the 25 repetitions; the two approaches were thus identical and are not listed here. In all other configuration scenarios, BasicILS found significantly better configurations than SimpleLS.

nomenon of overconfidence: optimizing performance using too small a training set leads to good training performance, but poor generalization to previously unseen test benchmarks. On the other hand, we clearly cannot evaluate every parameter configuration on an enormous training set—if we did, the search progress would be unreasonably slow.

To deal with this problem, we propose adaptively choosing the number of training instances to be used for the evaluation of each parameter configuration.

Definition 5 (Consistent estimator). *$\hat{c}_N(\theta)$ is a* consistent estimator *for $c(\theta)$ iff*

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

When $\hat{c}_N(\theta)$ is a consistent estimator of $c(\theta)$, cost estimates become more and more reliable as N approaches infinity, eventually eliminating overconfidence and the possibility of mistakes in comparing two parameter configurations (and thus, over-tuning). This fact is captured in the following lemma.

Lemma 6 (No mistakes for $N \to \infty$). *Let $\theta_1, \theta_2 \in \Theta$ be any two parameter configurations with $c(\theta_1) < c(\theta_2)$. Then, for consistent estimators $\hat{c}_N$, $\lim_{N\to\infty} P(\hat{c}_N(\theta_1) \geq \hat{c}_N(\theta_2)) = 0$.*

Proof. Write c_1 as shorthand for $c(\theta_1)$, c_2 for $c(\theta_2)$, $\hat{c}_1$ for $\hat{c}_N(\theta_1)$, and $\hat{c}_2$ for $\hat{c}_N(\theta_2)$. Define $m = \frac{1}{2}(c_2 + c_1)$ as the midpoint between c_1 and c_2, and $\epsilon = c_2 - m = m - c_1 > 0$ as its distance from each of the two points.

Since $\hat{c}_N$ is a consistent estimator for c, the estimate $\hat{c}_1$ comes arbitrarily close to the real cost c_1. That is, $\lim_{N\to\infty} P(|\hat{c}_1 - c_1| < \epsilon) = 1$. Since $|m - c_1| = \epsilon$, the estimate $\hat{c}_1$ cannot be greater than or equal to m: $\lim_{N\to\infty} P(\hat{c}_1 \geq m) = 0$. Similarly, $\lim_{N\to\infty} P(\hat{c}_2 < m) = 0$. Since

$$\begin{aligned} P(\hat{c}_1 \geq \hat{c}_2) &= P(\hat{c}_1 \geq \hat{c}_2 \wedge \hat{c}_1 \geq m) + P(\hat{c}_1 \geq \hat{c}_2 \wedge \hat{c}_1 < m) && (3) \\ &= P(\hat{c}_1 \geq \hat{c}_2 \wedge \hat{c}_1 \geq m) + P(\hat{c}_1 \geq \hat{c}_2 \wedge \hat{c}_1 < m \wedge \hat{c}_2 < m) && (4) \\ &\leq P(\hat{c}_1 \geq m) + P(\hat{c}_2 < m), && (5) \end{aligned}$$

we have $\lim_{N\to\infty} P(\hat{c}_1 \geq \hat{c}_2) \leq \lim_{N\to\infty} (P(\hat{c}_1 \geq m) + P(\hat{c}_2 < m)) = 0 + 0 = 0$. □

We note that in many practical scenarios cost estimators may not be consistent, i.e., may fail to closely approximate the true performance of a given parameter configuration even for a large number of runs of the target algorithm. For example, when a finite training set, Π, is used during configuration rather than a distribution over problem instances, $\mathcal{D}$, then for large N, $\hat{c}_N$ will only accurately reflect the cost of parameter configurations on the training set, Π. Even for large training sets, Π, this may subtly differ from the true cost as defined by performance across the whole distribution, $\mathcal{D}$; for small training sets the difference may be substantial, while it vanishes with training set size approaching infinity. We thus recommend the use of large training sets whenever possible.

6.1 The FocusedILS Algorithm

FocusedILS is a variant of ParamILS in which the number of training samples considered differs from one parameter configuration to another. We denote the number of runs available to estimate the cost statistic $c(\theta)$ for a parameter configuration θ by $N(\theta)$. Having performed different numbers of runs using different parameter configurations, we face the question of comparing two parameter configurations θ_1 and θ_2 for which $N(\theta_1) \neq N(\theta_2)$. One option would be simply to compute the empirical cost statistic based on the available number of runs for each configuration. However, considering the issues mentioned in the context of blocking on instances and seeds in Section 3.1, this may lead to systematic biases. For example, bias would be introduced if the first instances in our sampled list of $\langle$instance,seed$\rangle$ pairs are easier than the average instance, because parameter configurations with low numbers of runs would then tend to be preferred to configurations with higher numbers of runs. For this reason, when comparing two parameter configurations θ and θ' with $N(\theta) \leq N(\theta')$, we simply compare them based on the first $N(\theta)$ runs. Since the first $N(\theta)$ runs for both parameter configurations use exactly the same instances and seeds this allows us to compare quantities that can directly be compared without introducing bias or increasing variance. (This is similar to our strategy of blocking on instances and seeds, see Section 3.1.)

This approach to comparison leads us to the concept of domination.

Definition 7 (Domination)**.** θ_1 dominates θ_2 *if and only if* $N(\theta_1) \geq N(\theta_2)$ *and* $\hat{c}_{N(\theta_2)}(\theta_1) \leq \hat{c}_{N(\theta_2)}(\theta_2)$.

In words, θ_1 dominates θ_2 when at least as many runs have been conducted on θ_1 as on θ_2, and the performance of $\mathcal{A}(\theta_1)$ on the first $N(\theta_2)$ runs is at least as good as that of $\mathcal{A}(\theta_2))$ on all of its runs.

Now we are ready to discuss the comparison strategy encoded in procedure *better*$_{Foc}(\theta_1, \theta_2)$, which is used by the FocusedILS algorithm (see Procedure 4). This procedure first acquires one additional sample for the configuration with smaller $N(\theta_i)$ (or one for each configuration in the case of a tie). Then, it iteratively performs runs for the configuration with smaller $N(\theta_i)$ until one configuration dominates the other. At this point it returns true if θ_1 dominates θ_2 and false otherwise. We also keep track of the total number of runs performed since the last improving step (the last time *better*$_{Foc}$ returned true); we denote this number B. Whenever *better*$_{Foc}(\theta_1, \theta_2)$ returns true, we perform B "bonus" runs for θ_1 and reset B to 0. This mechanism ensures that we perform many runs with good configurations, and that the error made in every comparison of two configurations θ_1 and θ_2 decreases on expectation.

It is not difficult to show that in the limit, FocusedILS will sample every parameter configuration an unbounded number of times. The proof relies on the fact that, as an instance of ParamILS, FocusedILS performs random restarts with positive probability.

Procedure 4: better$_{Foc}(\theta_1, \theta_2)$
Procedure used in FocusedILS to compare two parameter configurations. Procedure $objective(\theta, N)$ returns the user-defined objective achieved by $\mathcal{A}_\theta$ on the first N instances, keeps track of the incumbent solution, and updates B (the number of runs performed since the last improvement step) as well as $\mathbf{R}_\theta$ (a global cache of algorithm runs performed with parameter configuration θ); it is detailed in Procedure 5 on page 31. For each θ, $N(\theta) = \text{length}(\mathbf{R}_\theta)$.

Input : Parameter configuration θ_1, parameter configuration θ_2
Output : True if θ_1 dominates θ_2, false otherwise

1 **if** $N(\theta_1) \leq N(\theta_2)$ **then** $\theta_{min} \leftarrow \theta_1; \theta_{max} \leftarrow \theta_2$
2 **else** $\theta_{min} \leftarrow \theta_2; \theta_{max} \leftarrow \theta_1$
3 **repeat**
4 $\quad i \leftarrow N(\theta_{min}) + 1$
5 $\quad \hat{c}_i(\theta_{max}) \leftarrow objective(\theta_{max}, i)$ *// If $N(\theta_{min}) = N(\theta_{max})$, adds a new run to $\mathbf{R}_{\theta_{max}}$.*
6 $\quad \hat{c}_i(\theta_{min}) \leftarrow objective(\theta_{min}, i)$ *// Adds a new run to $\mathbf{R}_{\theta_{min}}$.*
7 **until** *dominates(θ_1, θ_2) or dominates(θ_2, θ_1)*
8 **if** *dominates(θ_1, θ_2)* **then**
$\quad$ *// ===== Perform B bonus runs.*
9 $\quad \hat{c}_{N(\theta_1)+B}(\theta_1) \leftarrow objective(\theta_1, N(\theta_1) + B)$ *// Adds B new runs to $\mathbf{R}_{\theta_1}$.*
10 $\quad B \leftarrow 0$
11 $\quad$ **return** true
12 **else return** false
13 **Procedure** dominates(θ_1, θ_2)
14 **if** $N(\theta_1) < N(\theta_2)$ **then return** false
15 **return** $objective(\theta_1, N(\theta_2)) \leq objective(\theta_2, N(\theta_2))$

Lemma 8 (Unbounded number of evaluations)**.** *Let $N(J, \theta)$ denote the number of runs FocusedILS has performed with parameter configuration θ at the end of iteration J to estimate $c(\theta)$. Then, for any constant K and configuration $\theta \in \Theta$ (with finite Θ),* $\lim_{J\to\infty} [P(N(J, \theta)) \geq K] = 1$.

Proof. After each iteration of ParamILS, with probability $p_{restart} > 0$ a new configuration is picked uniformly at random, and with a probability of $1/|\Theta|$, this is configuration θ. The probability of visiting θ in an iteration is thus $p \geq \frac{p_{restart}}{|\Theta|} > 0$. Hence, the number of runs performed with θ is lower-bounded by a binomial random variable $\mathcal{B}(k; J, p)$. Then, for any constant $k < K$ we obtain $\lim_{J\to\infty} \mathcal{B}(k; J, p) = \lim_{J\to\infty} \binom{J}{k} p^k (1-p)^{J-k} = 0$. Thus, $\lim_{J\to\infty} [P(N(J, \theta)) \geq K] = 1$. □

Combining our two lemmata we can now show that in the limit, FocusedILS is guaranteed to converge to the true best parameter configuration.

Theorem 9 (Convergence of FocusedILS)**.** *When FocusedILS optimizes a cost statistic c based on a consistent estimator $\hat{c}_N$, the probability that it finds the true optimal parameter configuration θ^* approaches one as the number of iterations approaches infinity.*

Proof. According to Lemma 8, $N(\theta)$ grows unboundedly for each $\theta \in \Theta$. For each θ_1, θ_2, as $N(\theta_1)$ and $N(\theta_2)$ approach infinity, Lemma 6 states that in a pairwise comparison, the truly better

configuration will be preferred. Thus eventually, FocusedILS visits all finitely many parameter configurations and prefers the best one over all others with probability arbitrarily close to one. □

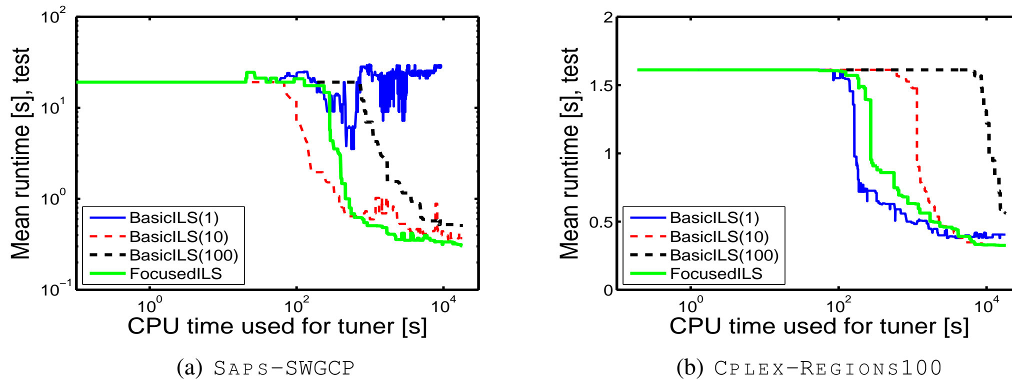
6.2 Experimental Evaluation of FocusedILS

In typical configuration scenarios, we expect that FocusedILS will sample good configurations much more frequently than bad ones, and therefore, that optimal configurations will be found much more quickly than is guaranteed by Theorem 9. In this section we investigate FocusedILS' performance experimentally.

In contrast to our previous comparison of RandomSearch, SimpleLS, and BasicILS using *training* performance, we now compare FocusedILS against BasicILS using *test* performance. This is motivated by the fact that while in our previous comparison all approaches used the same number of runs, N, to evaluate a parameter configuration, the number of runs FocusedILS uses grows over time. Even different repetitions of FocusedILS (using different training sets and random seeds) do not use the same number of runs to evaluate parameter configurations. However, they all eventually aim to optimize the same cost statistic c, and so test set performance (an unbiased estimator of c) provides a fairer basis for comparison than training performance. We only compare FocusedILS to BasicILS since BasicILS already outperformed RandomSearch and SimpleLS in Section 5.3.

Figure 10 compares the test performance of FocusedILS and BasicILS(N) with $N = 1$, 10, and 100. Using a single run to evaluate each parameter configuration, BasicILS(1) was fast but did not generalize well to the test set at all. Specifically, in configuration scenario SAPS-SWGCP BasicILS(1) selected a parameter configuration having test performance even worse than the default. On the other hand, using a large number of runs to evaluate each parameter configuration resulted in a very slow search but eventually led to parameter configurations with good test performance. We can observe this effect for BasicILS(100) in both scenarios in Figure 10. FocusedILS aims to achieve a fast search *and* good generalization to the test set. For the configuration scenarios in Figure 10, FocusedILS started quickly and also led to the best final performance.

We compare the performance of FocusedILS and BasicILS(100) for all configuration scenarios in Table 5. For two scenarios, FocusedILS performed statistically significantly better than BasicILS(100), and for a third it performed better with the difference falling just short of passing a significance test. These results are consistent with our past work in which we observed that FocusedILS always achieved statistically significantly better performance than BasicILS(100) (Hutter et al., 2007b). However, we found that in both configuration scenarios involving the SPEAR algorithm, BasicILS(100) actually performed better on average than FocusedILS, albeit not statistically significantly. We attribute this to the fact that for a complete, industrial solver such as SPEAR, the two benchmark distributions QCP and SWGCP are quite heterogeneous. This can already be seen by the variation in hardness between instances as shown in Figure 3. We expect FocusedILS to have problems in dealing with highly heterogeneous distributions due to the fact that it tries to extrapolate performance based on a few runs per parameter configuration. In combination with very large configuration spaces, such as encountered for SPEAR, this leads to a large number of parameter configurations being evaluated, but a large number of runs is performed for very few of these. In Section 8, we report experiments on a large number of further scenarios, and find that FocusedILS performs better than BasicILS in almost all of them.



(a) SAPS-SWGCP (b) CPLEX-REGIONS100

Figure 10: Test performance (mean runtime across 1,000 test instances) of BasicILS(N) with $N = 1$, $N = 10$, and $N = 100$ *vs* FocusedILS for two configuration scenarios. We plot median test performance over 25 repetitions of the configurators. Performance in the other three INDEPTH scenarios was qualitatively similar: BasicILS(1) was the fastest to move away from the starting parameter configuration, but its performance was not robust at all; BasicILS(10) was a rather good compromise between speed and generalization performance, but given enough time was outperformed by BasicILS(100). FocusedILS started finding good configurations quickly (except for scenario SPEAR-QCP, where it took even longer than BasicILS(100) to improve over the default) and always was amongst the best approaches at the end of the configuration process.

Scenario	Default	BasicILS(100)	FocusedILS	p-value
SAPS-SWGCP	20.41	0.59 ± 0.28	$\mathbf{0.32 \pm 0.08}$	$\mathbf{1.4 \times 10^{-4}}$
SPEAR-SWGCP	9.74	$\mathbf{8.13 \pm 0.95}$	8.40 ± 0.92	(0.21)
SAPS-QCP	9.80	5.48 ± 0.5	$\mathbf{5.20 \pm 0.38}$	0.08
SPEAR-QCP	2.65	$\mathbf{1.32 \pm 0.34}$	1.35 ± 0.20	(0.66)
CPLEX-REGIONS100	1.61	0.72 ± 0.45	$\mathbf{0.33 \pm 0.03}$	$\mathbf{1.2 \times 10^{-5}}$

Table 5: Comparison of test performance (mean runtime over 1,000 test instances, in CPU seconds) for BasicILS(100) and FocusedILS. For each configuration scenario, we report test performance of the default parameter configuration, mean $\pm$ stddev of the test performance reached by 25 repetitions of BasicILS(100) and FocusedILS, and the p-value for a paired Max-Wilcoxon test (see Section 3.1.3) for the difference of the two configurator's performance.

7. Adaptive Capping of Algorithm Runs

Now we consider the last of our dimensions of automated algorithm configuration, the cutoff time for each run of the target algorithm. We introduce a novel, effective and simple capping technique that adaptively determines the cutoff time for each run. The motivation for this capping technique comes from a problem encountered by all configuration procedures considered in this article (RandomSearch, SimpleLS, BasicILS, and FocusedILS): sometimes the search for a performance-optimizing parameter setting spends a lot of time with evaluating a parameter configuration that is much worse than other, previously seen configurations.

Consider, for example, a case where parameter configuration θ_1 takes a total of 10 seconds to solve each of 100 instances (i.e., it has a mean runtime of 0.1 seconds per instance), and another parameter configuration θ_2 takes one second per instance, i.e., 100 seconds for solving all instances. In order to compare θ_1 and θ_2 based on the above 100 instances, knowing the result for θ_1, it is not necessary to run θ_2 on all 100 instances. After 11 instances θ_2's mean runtime is already lower-bounded by $11/100 = 0.11$ seconds, because the remaining 89 instances could take no less than zero time. Since this lower bound exceeds the mean runtime of θ_1, we can be certain that the comparison will favour θ_1 without performing any of the remaining 89 runs. This effect provides the basis for our *adaptive capping* technique.

We first introduce a trajectory-preserving version of adaptive capping (**TP capping**) that provably does not change the search trajectory, but can still lead to large computational savings. For BasicILS and FocusedILS, we also modify this strategy heuristically to perform more aggressive adaptive capping (**Aggr capping**) and yield even better performance in practice. (For RandomSearch and SimpleLS, TP capping and Aggr capping are identical.)

7.1 Adaptive Capping for RandomSearch and SimpleLS

The simplest cases for our adaptive capping technique are RandomSearch and SimpleLS. Whenever these configurators evaluate a new parameter configuration θ, they perform a comparison *better*$_N(\theta, \theta_{inc})$ (for RandomSearch, see Algorithm 1 on page 13; for SimpleLS, see procedure *IterativeFirstImprovement* in the general ParamILS framework in Algorithm 2 on page 22). Without adaptive capping, these comparisons can take a long time, since a poor parameter configuration can easily take more than an order of magnitude longer than good configurations (see Section 4.2).

For the case of optimizing the mean of some non-negative cost (such as runtime or solution cost), we implement a bounded evaluation of a parameter configuration θ based on N runs and a given performance bound in Procedure $objective$ (see Procedure 5). This procedure sequentially performs runs for parameter configuration θ and after each run computes a lower bound on $\hat{c}_N(\theta)$ based on the $i \leq N$ runs performed so far. Specifically, for our objective of mean runtime we sum the runtimes of each of the i runs, and divide this sum by N; since all runtimes must be nonnegative, this quantity lower-bounds $\hat{c}_N(\theta)$. Once the lower bound exceeds the bound passed as an argument, we can skip the remaining runs for θ.

In order to pass the appropriate bounds to Procedure $objective$, we need to slightly modify Procedure *better*$_N$ (see Procedure 3 on page 22) for adaptive capping. Procedure $objective$ now has a bound as an additional third argument, which has to be set to ∞ in line 1 of *better*$_N$, and to $\hat{c}_N(\theta_2)$ in line 2. (Note that the reason for computing $\hat{c}_N(\theta_2)$ before $\hat{c}_N(\theta_1)$ is that in RandomSearch, θ_2 is always the incumbent solution, i.e., the best configuration encountered so far, and we thus bound the evaluation of any new parameter configuration by $\hat{c}_N(\theta_{inc})$).

Procedure 5: $objective(\theta, N,$ optional parameter $bound)$
Procedure that computes $\hat{c}_N(\theta)$, either by performing new runs or by exploiting previous cached runs. An optional third parameter specifies a bound on the computation to be performed; when this parameter is not specified, the bound is taken to be ∞. For each θ, $N(\theta)$ is the number of runs performed for θ, i.e., the length of the global array $\mathbf{R}_\theta$. When computing runtimes, we count unsuccessful runs as taking 10 times their cutoff time.

Input : Parameter configuration θ, number of runs, N, optional bound $bound$
Output : $\hat{c}_N(\theta)$ if $\hat{c}_N(\theta) \leq bound$, otherwise a large constant (maxPossibleObjective) plus the number of instances that remain unsolved when the bound was exceeded
Side Effect: Adds runs to the global cache of performed algorithm runs, $\mathbf{R}_\theta$; updates counter of bonus runs, B

// ===== Maintain invariant: $N(\theta_{inc}) \geq N(\theta)$ for any θ
1 **if** $\theta \neq \theta_{inc}$ *and* $N(\theta_{inc}) < N$ **then**
2 $\quad$ $\hat{c}_N(\theta_{inc}) \leftarrow objective(\theta_{inc}, N, \infty)$ *// Adds $N - N(\theta_{inc})$ runs to $\mathbf{R}_{\theta_{inc}}$*
// ===== For aggressive capping, update bound.
3 **if** *Aggressive capping* **then** $bound \leftarrow \min(bound, bm \times \hat{c}_N(\theta_{inc}))$
// ===== Update the run results in tuple $\mathbf{R}_\theta$.
4 **for** $i = 1...N$ **do**
5 $\quad$ sum_runtime $\leftarrow$ sum of runtimes in $\mathbf{R}_\theta[1], \ldots, \mathbf{R}_\theta[i-1]$ *// Tuple indices starting at 1.*
6 $\quad$ $\kappa'_i \leftarrow \max(\kappa, N \times bound -$ sum_runtime$)$
7 $\quad$ **if** $N(\theta) \geq i$ **then** $(\theta, \pi_i, \kappa_i, o_i) \leftarrow \mathbf{R}_\theta[i]$
8 $\quad$ **if** $N(\theta) \geq i$ **and** *(($\kappa_i \geq \kappa'_i$* **and** *$o_i =$ "unsuccessful")* **or** *($\kappa_i < \kappa'_i$* **and** *$o_i \neq$ "unsuccessful"))* **then**
9 $\quad\quad$ $o'_i \leftarrow o_i$ *// Previous run is longer yet unsuccessful or shorter yet successful ⇒ can re-use result*
10 $\quad$ **else**
11 $\quad\quad$ $o'_i \leftarrow$ objective from a newly executed run of $\mathcal{A}_\theta$ on instance π_i with seed s_i and captime κ_i
12 $\quad\quad$ $B \leftarrow B + 1$ *// For FocusedILS, keep track of number of runs since last improving step.*
13 $\quad$ $\mathbf{R}_\theta[i] \leftarrow (\theta, \pi_i, \kappa'_i, o'_i)$
14 $\quad$ **if** $1/N \times (sum_runtime + o'_i) > bound$ **then return** maxPossibleObjective + $(N+1) - i$
15 **if** $N = N(\theta_{inc})$ *and (sum of runtimes in $\mathbf{R}_\theta$) < (sum of runtimes in $\mathbf{R}_{\theta_{inc}}$)* **then** $\theta_{inc} \leftarrow \theta$
16 **return** $1/N \times$ (sum of runtimes in $\mathbf{R}_\theta$)

Although in this work we focus on the objective of minimizing mean runtime for decision algorithms, we note that our adaptive capping technique can be applied to other configuration objectives. This is straightforward in the case of any other objective that is based on a a mean (e.g., mean solution quality). It also works for other configuration objectives, in particular for quantiles, such as the median or the 90% quantile, which we have considered in previous work. In the case of the median, with $M \leq N/2$ runs, we can only obtain a trivial bound of zero; however, for $M > N/2$, the $N/2$th-worst encountered cost of the M runs provides a lower bound on $\hat{c}_N(\theta)$. A similar result holds for other quantiles. Interestingly, the more emphasis is put on robustness, the better for our adaptive capping technique. When the Q% runtime quantile is to be minimized, bounds become non-trivial after observing the performance for $M > N/(100 - Q)$ runs. For example, after 11 timed-out runs using a configuration θ, it can be concluded that $\theta's$ 90% quantile based on 100 runs

will be larger than that of a parameter configuration that did not show any timeouts for 100 runs with the same cutoff time.

7.2 Adaptive Capping in BasicILS

Now, we consider the application of adaptive capping to BasicILS.

7.2.1 Trajectory-preserving Capping

In essence, adaptive capping for BasicILS can be implemented by applying the same changes to Procedure $better_N$ as described above for RandomSearch. However, unlike in the case of RandomSearch, for BasicILS, the current configuration is not always compared to the incumbent configuration, but often to the best encountered *in a given iteration* of the ILS procedure, which we refer to as the *current iteration's best configuration*. Therefore, where in RandomSearch we compared lower bounds on solution costs with the performance of the incumbent configuration, we now compare them to the performance of the current iteration's best configuration. Since during most of the search process, the current iteration's best configuration is typically quite good, significant time can be saved by using adaptive capping. Because this approach amounts to computing exactly the same function $better_N$ as used in the original version of BasicILS, the modified procedure follows exactly the same search trajectory it would have followed without capping, but typically requires much less runtime. Hence, within the same amount of overall running time, this new version of BasicILS tends to be able to search a larger part of the parameter configuration space.

7.2.2 Aggressive Capping

While the use of trajectory-preserving adaptive capping can result in substantial speedups of BasicILS, it is not always equally effective. One key reason for this comes from the fact that configurations are not always compared to the incumbent, but sometimes to significantly worse-performing configurations. In particular, this happens after perturbation phases of the ILS procedure. Since a new iteration starts after each perturbation, the current iteration's best configuration right after a perturbation is simply the parameter configuration resulting from the perturbation. In the frequent case that that configuration performs poorly, the capping criterion does not apply as quickly as when the comparison is performed against a high-performing configuration.

To counteract this effect, we introduced a more aggressive capping strategy that may terminate the evaluation of a poor performing configuration at any time. In this heuristic extension of our adaptive capping technique, we bound the evaluation of *any* parameter configuration by the performance of the incumbent parameter configuration multiplied by a factor that we call the *bound multiplier*, bm. When a comparison between any two parameter configurations θ and θ' is performed and the evaluations of both are terminated preemptively, the configuration having solved more instances within the allowed time is taken to be the better one. (This behaviour is achieved by line 14 in Procedure $objective$ that keeps track of the number of instances solved when exceeding the bound.) Ties are broken to favour moving to a new parameter configuration instead of staying with the current one.

Depending on the bound multiplier, the use of this aggressive capping mechanism may change the search trajectory of BasicILS. While for $bm = \infty$, the heuristic method reduces to our trajectory-preserving method, a very aggressive setting of $bm = 1$ means that once we know a parameter configuration to be worse than the incumbent, we stop its evaluation. In our experiments we set

$bm = 2$, that is, once the lower bound on the performance of a configuration exceeds twice the performance of the incumbent solution, its evaluation is terminated. (In Section 9, we revisit this choice of $bm = 2$, configuring the parameters of ParamILS itself.)

7.3 Adaptive Capping in FocusedILS

The main difference between BasicILS and FocusedILS is that the latter adaptively varies the number of runs used to evaluate each parameter configuration. This difference complicates, but does not rule out, the use of adaptive capping. This is because FocusedILS always compares pairs of parameter configurations based on the same number of runs for each configuration, even though this number can differ from one comparison to the next.

Thus, we can extend adaptive capping to FocusedILS by using separate bounds for every number of runs, N. Recall that FocusedILS never moves from one configuration, θ, to a neighbouring configuration, θ', without performing at least as many runs for θ' as have been performed for θ. Since we keep track of the performance of θ with any number of runs $M \leq N(\theta)$, a bound for the evaluation of θ' is always available. Therefore, we can implement both, trajectory-preserving and aggressive capping as we did for BasicILS.

Equivalently to BasicILS, for FocusedILS the inner workings of adaptive capping are implemented in Procedure $objective$ (see Procedure 5). We only need to modify Procedure $better_{Foc}$ (see Procedure 4 on page 27) to call $objective$ with the right bounds. This leads to the following changes in Procedure $better_{Foc}$: subprocedure $dominates$ on line 13 now takes a bound as an additional argument and passes it on to the two calls to $objective$ in line 15. The two calls of $dominates$ in line 7 and the one call in line 8 all use the bound $\hat{c}_{\theta_{max}}$. The three direct calls to $objective$ in lines 5, 6, and 9 use bounds ∞, $\hat{c}_{\theta_{max}}$, and ∞, respectively.

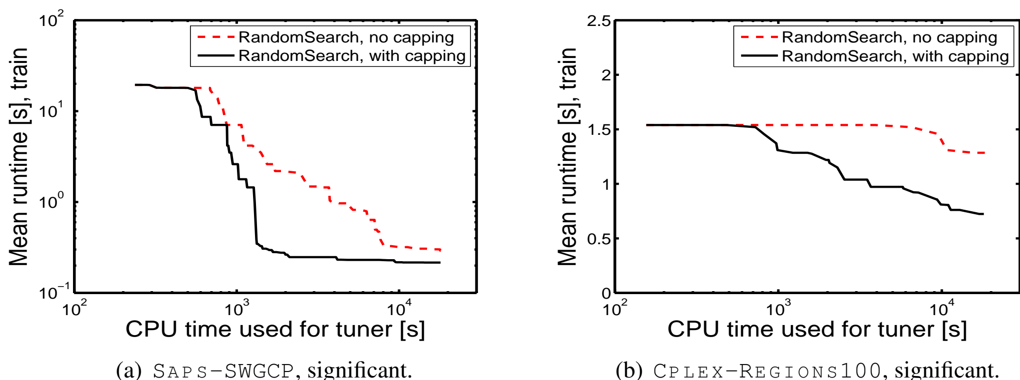
7.4 Experimental Evaluation of Adaptive Capping

We now present experimental evidence that the use of adaptive capping has a strong impact on the performance of RandomSearch, BasicILS and FocusedILS.

Figure 11 compares training performance of RandomSearch on two configuration scenarios with and without adaptive capping, and Table 6 quantifies the speedups for all INDEPTH scenarios. Overall, adaptive capping enabled RandomSearch to evaluate up to 30 times as many parameter configurations in the same time, while preserving the search trajectory (given the same random seeds in both cases). This enabled RandomSearch to find parameter configurations with statistically significantly better training performance in all of our configuration scenarios.

Figure 12 illustrates the extent to which TP capping sped up BasicILS in the same two configuration scenarios. In both cases, capping helped to improve training performance substantially; for SAPS-SWGCP, BasicILS found the same solutions up to about an order of magnitude faster than without capping. Table 7 quantifies the speedups for all five INDEPTH configuration scenarios. TP capping enabled up to four times as many iterations (in SAPS-SWGCP) and improved average performance in all scenarios. The improvement was statistically significant in three of the five scenarios.

Aggressive capping for BasicILS was found to further improve performance in several scenarios. In the first iteration, both capping techniques are identical (the best configuration in that iteration is always the incumbent). Thus, we did not observe a difference on configuration scenarios SPEAR-SWGCP and CPLEX-REGIONS100, for which none of the 25 repetitions of the configurator finished its first iteration. Table 8 shows results for the other three configuration scenarios. Note that



(a) SAPS-SWGCP, significant.

(b) CPLEX-REGIONS100, significant.

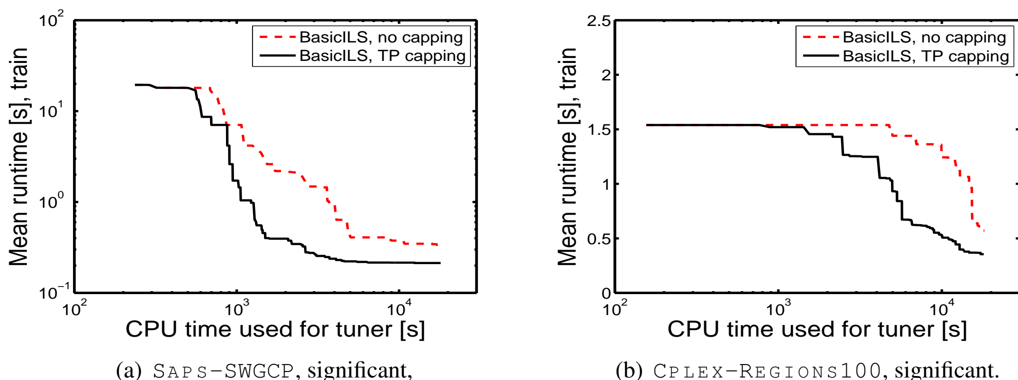
Figure 11: Speedup of RandomSearch by adaptive capping for two configuration scenarios. We performed 25 repetitions of RandomSearch(100) with and without adaptive capping. For each time step, we computed training performance of each repetition (mean runtime over $N = 100$ training instances) and plot the median over the 25 repetitions.

Scenario	Mean runtime of best configuration			Avg. # configurations evaluated	
	No capping	TP/Aggr capping	p-value	No capping	TP/Aggr capping
SAPS-SWGCP	0.46 ± 0.34	$\mathbf{0.21 \pm 0.03}$	$\mathbf{1.2 \times 10^{-5}}$	60	**2004**
SPEAR-SWGCP	7.02 ± 1.11	$\mathbf{6.70 \pm 1.20}$	$\mathbf{6.1 \times 10^{-5}}$	71	**199**
SAPS-QCP	3.73 ± 1.53	$\mathbf{3.40 \pm 1.53}$	$\mathbf{4.0 \times 10^{-5}}$	127	**434**
SPEAR-QCP	0.58 ± 0.59	$\mathbf{0.45 \pm 0.51}$	$\mathbf{6.0 \times 10^{-5}}$	321	**1951**
CPLEX-REGIONS100	1.29 ± 0.28	$\mathbf{0.70 \pm 0.12}$	$\mathbf{1.8 \times 10^{-5}}$	45	**1004**

Table 6: Effect of adaptive capping on training performance (mean runtime on $N = 100$ training instances, in CPU seconds) of RandomSearch. For each configuration scenario, we report mean $\pm$ stddev of the training performance reached by 25 repetitions of the configurator with and without adaptive capping (for RandomSearch, TP and Aggr capping are identical), the p-value for a paired Max-Wilcoxon test (see Section 3.1.3) for the difference between the two, and the average number of configurations they evaluated. Note that the speedup is highly significant in all scenarios.

the number of iterations for configuration scenario SAPS-SWGCP increased from 12 to 219, and that for this scenario the gains in training performance were also statistically significant. For the other configuration scenarios, the gains were less pronounced.

Since adaptive capping improved RandomSearch more than BasicILS, one might now wonder how our previous comparison between these approaches changes when adaptive capping is used. We found that the gap between the techniques narrowed, but that the qualitative differences persisted. Specifically, Table 9 compares the training performance of RandomSearch(100) and BasicILS(100) with adaptive capping enabled. Comparing this to the performance differences without capping (see Table 3), now BasicILS only performed significantly better in two of the five INDEPTH scenarios (as



(a) SAPS-SWGCP, significant,

(b) CPLEX-REGIONS100, significant.

Figure 12: Speedup of BasicILS by adaptive capping for two configuration scenarios. We performed 25 repetitions of BasicILS(100) without adaptive capping and with TP capping. For each time step, we computed training performance of each repetition (mean runtime over $N = 100$ training instances) and plot the median over the 25 repetitions.

Scenario	Mean runtime of best configuration			Avg. # iterations	
	No capping	TP capping	p-value	No capping	TP capping
SAPS-SWGCP	0.38 ± 0.19	$\mathbf{0.24 \pm 0.05}$	$\mathbf{6.1 \times 10^{-5}}$	3	**12**
SPEAR-SWGCP	6.78 ± 1.73	$\mathbf{6.65 \pm 1.48}$	$\mathbf{0.01}$	1	1
SAPS-QCP	2.86 ± 1.25	$\mathbf{2.83 \pm 1.27}$	0.31	5	**10**
SPEAR-QCP	0.361 ± 0.39	$\mathbf{0.356 \pm 0.44}$	0.66	2	**3**
CPLEX-REGIONS100	0.67 ± 0.35	$\mathbf{0.47 \pm 0.26}$	$\mathbf{7.3 \times 10^{-4}}$	1	1

Table 7: Effect of adaptive capping on training performance (mean runtime on $N = 100$ training instances, in CPU seconds) for BasicILS(100). For each configuration scenario, we report mean $\pm$ stddev of the final training performance reached by 25 repetitions of the configurator capping and with TP capping, the p-value for a paired Max-Wilcoxon test for their difference (see Section 3.1.3), as well as their average number of iterations.

opposed to three without capping). However, BasicILS still performed better than RandomSearch in all configuration scenarios on average.

We now move to the experimental evaluation of capping for FocusedILS. Training performance is not a useful quantity in the context of comparing different versions of FocusedILS, since the number of runs this measure is based on varies widely between repetitions of the configurator. Instead, we used two other measures to quantify search progress: the number of iterations performed and the number $N(\theta_{inc})$ of runs performed for the incumbent parameter configuration. Table 10 shows these two measures for our five INDEPTH configuration scenarios and the three capping schemes (none, TP, Aggr). FocusedILS with TP capping achieved higher values than without capping for all scenarios and both measures (although only some of the differences were statistically significant). Aggressive capping increased both measures further for all scenarios, and most of the differences between no capping and aggressive capping were statistically significant.

Scenario	Mean runtime of best configuration			Avg. # iterations	
	TP capping	Aggr capping	p-value	TP capping	Aggr capping
SAPS-SWGCP	0.24 ± 0.05	$\mathbf{0.21 \pm 0.03}$	$\mathbf{1.1 \times 10^{-4}}$	12	**219**
SAPS-QCP	2.83 ± 1.27	$\mathbf{2.78 \pm 1.28}$	0.13	10	**11**
SPEAR-QCP	0.356 ± 0.44	$\mathbf{0.356 \pm 0.41}$	0.37	3	3

Table 8: Comparison of training performance (mean runtime over $N = 100$ training instances, in CPU seconds) for TP capping and Aggr capping ($bm = 2$) in BasicILS(100). In configuration scenarios SPEAR-SWGCP and CPLEX-REGIONS100, BasicILS only performed one iteration; therefore, the two capping approaches were identical and are not listed here. For the other scenarios, we report mean $\pm$ stddev of the training performance reached by 25 repetitions of the configurator with TP and Aggr capping, the p-value for a paired Max-Wilcoxon test for their difference (see Section 3.1.3), as well as their average number of iterations.

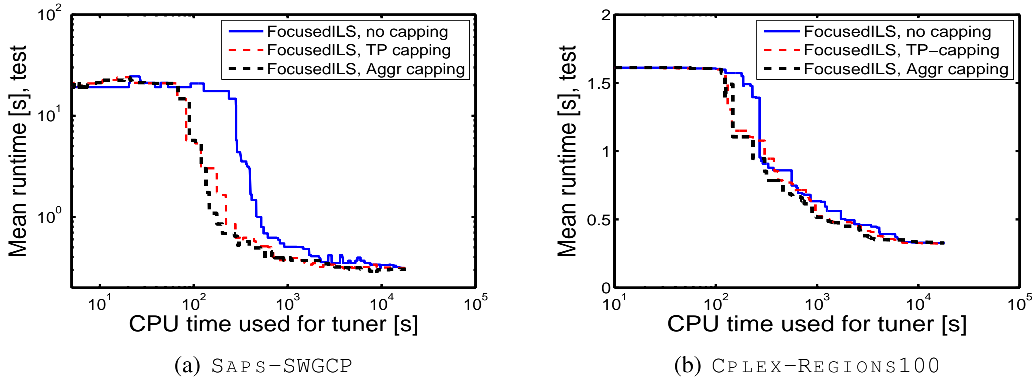
Scenario	RandomSearch(100)	BasicILS(100)	p-value
SAPS-SWGCP	0.215 ± 0.034	$\mathbf{0.214 \pm 0.034}$	0.35
SPEAR-SWGCP	6.70 ± 1.20	$\mathbf{6.65 \pm 1.48}$	0.51
SAPS-QCP	3.40 ± 1.53	$\mathbf{2.78 \pm 1.28}$	$\mathbf{2.7 \times 10^{-5}}$
SPEAR-QCP	0.45 ± 0.51	$\mathbf{0.36 \pm 0.41}$	0.28
CPLEX-REGIONS100	0.73 ± 0.1	$\mathbf{0.47 \pm 0.26}$	**0.0013**

Table 9: Comparison of final training performance (mean runtime over 100 training instances, in CPU seconds) of BasicILS(100) and RandomSearch(100), both using the same $N = 100$ instances, with capping (aggressive capping with $bm = 2$; the equivalent table without capping is Table 3.) We also list the p-value for a paired Max-Wilcoxon test (see Section 3.1.3) for the difference of the two configurators' performance.

Figure 13 demonstrates that for two configuration scenarios FocusedILS with capping reached the same solution qualities more quickly than without capping. After finding the respective configurations, the performance of FocusedILS shows no further significant increase. The statistics in Table 11 confirm that at the end of the run there is no statistically significant difference between FocusedILS with and without capping in any of our five INDEPTH configuration scenarios. The same table shows that in BasicILS, capping *does* lead to statistically significant improvements in the two scenarios SAPS-SWGCP and CPLEX-REGIONS100, for which FocusedILS clearly outperforms BasicILS when both are ran without capping. Comparing the performance of FocusedILS and BasicILS(100) with capping in Table 11, we find that FocusedILS is only significantly better for scenario CPLEX-REGIONS100 (which is in some sense the hardest), with insignificant differences for the other scenarios. This and the stagnation effect seen in Figure 13 leads us to believe that further improvements are hard to obtain for these configuration scenarios, but that capping consistently helps the ParamILS variants to find good parameter configurations more quickly.

Number of iterations performed					
Scenario	No capping	TP capping	p-value	Aggr capping	p-value
SAPS-SWGCP	121 ± 12	166 ± 15	$\mathbf{1.2 \times 10^{-5}}$	$\mathbf{244 \pm 19}$	$\mathbf{1.2 \times 10^{-5}}$
SPEAR-SWGCP	37 ± 12	43 ± 15	$\mathbf{0.0026}$	$\mathbf{47 \pm 18}$	$\mathbf{9 \times 10^{-5}}$
SAPS-QCP	150 ± 31	150 ± 28	0.94	$\mathbf{155 \pm 31}$	0.32
SPEAR-QCP	153 ± 49	165 ± 41	$\mathbf{0.03}$	$\mathbf{213 \pm 62}$	$\mathbf{1.2 \times 10^{-5}}$
CPLEX-REGIONS100	36 ± 13	40 ± 16	0.26	$\mathbf{54 \pm 15}$	$\mathbf{1.8 \times 10^{-5}}$
Number of runs $N(\theta_{inc})$ performed for the incumbent parameter configuration					
Scenario	No capping	TP capping	p-value	Aggr capping	p-value
SAPS-SWGCP	993 ± 211	1258 ± 262	$\mathbf{4.7 \times 10^{-4}}$	$\mathbf{1818 \pm 243}$	$\mathbf{1.2 \times 10^{-5}}$
SPEAR-SWGCP	503 ± 265	476 ± 238	(0.58)	$\mathbf{642 \pm 288}$	$\mathbf{0.009}$
SAPS-QCP	1702 ± 397	1748 ± 383	0.40	$\mathbf{1777 \pm 306}$	0.17
SPEAR-QCP	836 ± 509	1130 ± 557	$\mathbf{0.02}$	$\mathbf{1215 \pm 501}$	$\mathbf{0.003}$
CPLEX-REGIONS100	761 ± 215	795 ± 184	0.40	$\mathbf{866 \pm 232}$	0.07

Table 10: Effect of adaptive capping on search progress in FocusedILS, as measured by the number of iterations performed and the number of runs $N(\theta_{inc})$ performed for the incumbent parameter configuration. For each configuration scenario, we report mean $\pm$ stddev of both of these measures across 25 repetitions of the configurator without capping, with TP capping, and with Aggr capping, as well as the p-values for paired Max-Wilcoxon tests (see Section 3.1.3) for the differences between no capping and TP capping; and between no capping and Aggr capping.



(a) SAPS-SWGCP (b) CPLEX-REGIONS100

Figure 13: Speedup of FocusedILS by adaptive capping for two configuration scenarios. We performed 25 repetitions of FocusedILS without adaptive capping, with TP capping and with Aggr capping. For each time step, we computed the test performance of each repetition (mean runtime over $1,000$ test instances) and plot the median over the 25 repetitions. The differences at the end of the trajectory were not statistically significant; however, with capping the time required to achieve that quality was lower in these two configuration scenarios. In the other three scenarios, the gains due to capping were smaller.

FocusedILS					
Scenario	No capping	TP capping	p-value	Aggr capping	p-value
SAPS-SWGCP	0.325 ± 0.08	0.326 ± 0.06	(0.81)	$\mathbf{0.316 \pm 0.05}$	0.72
SPEAR-SWGCP	8.41 ± 0.92	8.49 ± 1.07	(0.48)	$\mathbf{8.3 \pm 1.06}$	0.68
SAPS-QCP	5.2 ± 0.38	$\mathbf{5.12 \pm 0.31}$	0.47	5.21 ± 0.39	(0.72)
SPEAR-QCP	1.35 ± 0.2	$\mathbf{1.25 \pm 0.21}$	0.19	1.29 ± 0.2	0.28
CPLEX-REGIONS100	0.333 ± 0.03	$\mathbf{0.326 \pm 0.01}$	0.24	0.346 ± 0.04	(0.46)
BasicILS(100)					
Scenario	No capping	TP capping	p-value	Aggr capping	p-value
SAPS-SWGCP	0.59 ± 0.28	0.36 ± 0.09	$\mathbf{2.5 \times 10^{-4}}$	$\mathbf{0.323 \pm 0.06}$	$\mathbf{5 \times 10^{-5}}$
SPEAR-SWGCP	8.13 ± 0.95	$\mathbf{8.05 \pm 0.9}$	0.25	identical to TP capping	
SAPS-QCP	5.48 ± 0.5	$\mathbf{5.44 \pm 0.51}$	0.5	5.5 ± 0.53	(1)
SPEAR-QCP	$\mathbf{1.32 \pm 0.34}$	1.33 ± 0.37	(0.58)	1.39 ± 0.33	(0.24)
CPLEX-REGIONS100	0.72 ± 0.45	$\mathbf{0.5 \pm 0.3}$	$\mathbf{7.3 \times 10^{-4}}$	identical to TP capping	

Table 11: Effect of adaptive capping on test performances (mean runtime over $1,000$ test instances, in CPU seconds) of FocusedILS and BasicILS(100). For each configuration scenario and procedure, we report mean $\pm$ stddev of the test performance reached by 25 repetitions of the configurator without capping, with TP capping, and with Aggr capping, as well as the p-values for paired Max-Wilcoxon tests (see Section 3.1.3) for the differences in performances between no capping and TP capping; and between no capping and Aggr capping. Comparing FocusedILS and BasicILS(100) (both with Aggr capping), FocusedILS is significantly better for scenario CPLEX-REGIONS100 (p-value 0.008), insignificantly better for scenarios SAPS-SWGCP, SAPS-QCP, and SPEAR-QCP (p-values 0.67, 0.07, and 0.53), and insignificantly worse for scenario SPEAR-SWGCP (p-value 0.18).

8. Experiments for Broad Collection of Configuration Scenarios

In this section, we present results for automatically configuring SAPS, SPEAR, and CPLEX on benchmark sets that go beyond the three we have considered so far. We first introduce the new configuration scenarios, briefly explain our experimental setup and then report the results of our study.

8.1 Broad Configuration Scenarios

We constructed a broad range of configuration scenarios by acquiring interesting instances from public benchmark libraries and other researchers. These configuration scenarios include two new scenarios each for SAPS and SPEAR and five for CPLEX; they are summarized in Table 12. The target algorithms for our BROAD scenarios are the same as we have used throughout the paper (and described in Section 3.2.1). Here, we give some details about the new instance benchmark sets; all of these sets were split 50:50 into disjoint training and test sets.

SWV This set of SAT-encoded software verification instances comprises 604 instances generated by the CALYSTO static checker (Babić and Hu, 2007). It was generated by Domagoj Babić and was also used in a previous application study of ParamILS (Hutter et al., 2007a). These instances contained an average of $64,416$ variables and $195,058$ clauses, with respective standard deviations of $53,912$ and $174,534$.

Scenario	Type of benchmark instances & citation	# training	# test
SPEAR-SWV	SAT-encoded software verification (Babić and Hu, 2007)	302	302
SPEAR-IBM	SAT-encoded bounded model checking (Zarpas, 2005)	382	383
SAPS-RANDOM	SAT competition instances, random category (see, e.g., Le Berre and Simon, 2004)	363	363
SAPS-CRAFTED	SAT competition instances, crafted category (see, e.g., Le Berre and Simon, 2004)	189	188
CPLEX-REGIONS200	Combinatorial Auctions (CATS) (Leyton-Brown et al., 2000)	1,000	1,000
CPLEX-CONIC.SCH	Machine-Job Assignment (BCOL) (Aktürk et al., 2007)	172	171
CPLEX-CLS	Capacitated Lot Sizing (BCOL) (Atamtürk and Muñoz, 2004)	50	50
CPLEX-MIK	Mixed-integer knapsack (BCOL) (Atamtürk, 2003)	60	60
CPLEX-QP	Quadratic programs from RNA energy parameter optimization (Andronescu et al., 2007)	1,000	1,000

Table 12: Overview of our nine BROAD configuration scenarios. BCOL stands for Berkeley Computational Optimization Lab, CATS for Combinatorial Auction Test Suite.

IBM This set of SAT-encoded bounded model checking instances comprises 765 instances generated by Zarpas (2005), and was also used in the above-mentioned application study of ParamILS (Hutter et al., 2007a). These instances contained an average of $91,041$ variables and $386,171$ clauses, with respective standard deviations of $149,815$ and $646,813$. Some of the instances in this set are extremely hard, the largest instance containing $1,400,478$ variables and $5,502,329$ clauses. In order to reduce training time, we removed the 95 instances from the training set that could not be solved by SPEAR's default parameter configuration within one hour, leaving 287 instances for training (this was identical to our methodology in Hutter et al. (2007a)).

Random This set comprises 726 satisfiable instances collected from the RANDOM categories of the SAT competitions 2003, 2004, 2005, and 2007[3] (see, e.g., Le Berre and Simon, 2004). This instance set is very heterogeneous and contains an average of $1,004$ variables and $6,707$ clauses, with respective standard deviations of $1,859$ and $8,236$.

Crafted This set comprises 377 satisfiable instances collected from the CRAFTED (or HANDMADE) categories of the SAT competitions 2003, 2004, 2005, and 2007 (see, e.g., Le Berre and Simon, 2004). These instances were contributed by a variety of researchers and are very heterogeneous. They contain an average of $2,319$ variables and $48,914$ clauses, with respective standard deviations of $5,801$ and $183,632$. The largest instance contains $75,600$ variables and $2,340,195$ clauses.

Regions200 This set is almost identical to the **Regions100** set used throughout the paper, but is much larger. We generated 2,000 MILP instances with the generator provided with the Combinatorial Auction Test Suite (Leyton-Brown et al., 2000), based on the `regions` option with the goods parameter set to 200 and the bids parameter set to 1,000. These instances contain an average of $1,002$ variables and 385 inequalities, with respective standard deviations of 1.7 and 3.4.

CONIC.SCH This set comprises 343 machine-job assignment instances encoded as mixed integer quadratically constrained programs (MIQCP). It was obtained from the Berkeley Computational

3. In 2006 a "SAT Race" was run instead of a "SAT competition,"; the SAT Race considers only industrial instances.

Optimization Lab[4] and was introduced by Aktürk et al. (2007). These instances contain an average of $2,769$ variables and 2255 constraints, with respective standard deviations of $2,133$ and $1,592$.

CLS This set comprises 100 capacitated lot-sizing instances encoded as mixed integer linear programs (MILP). It was also obtained from the Berkeley Computational Optimization Lab and was introduced by Atamtürk and Muñoz (2004). All 100 instances contain 181 variables and 180 constraints.

MIK This set of 120 MILP-encoded mixed-integer knapsack instances was also obtained from the Berkeley Computational Optimization Lab and was originally introduced by Atamtürk (2003). These instances contain an average of 384 variables and 151 constraints, with respective standard deviations of 309 and 127.

QP This set of quadratic programs originated from RNA energy parameter optimization (Andronescu et al., 2007). Mirela Andronescu generated 2,000 instances for our experiments. These instances contain $9,366$ variables and $9,191$ constraints on average, with respective standard deviations of $7,165$ and $7,186$. Since the instances are polynomial-time solvable quadratic programs, we set a large number of inconsequential CPLEX parameters concerning the branch and cut mechanism to their default values, ending up with 27 categorical, 2 integer and 2 continuous parameters to be configured, for a discretized parameter configuration space of size 3.27×10^{17}.

8.2 Experimental Setup

For the BROAD configuration scenarios considered in this section, we set significantly longer cutoff times than on the INDEPTH configuration scenarios we studied previously, in order to study ParamILS's behavior when faced with harder problems. Specifically, we used a cutoff time of 300 CPU seconds during training and allowed a running time of two CPU days per configurator and repetition. As always, our configuration objective was to minimize mean runtime, and we counted timeouts as ten times the cutoff time (in this case, $3,000$ seconds). We employed our final versions of BasicILS(100) and FocusedILS, both with aggressive capping ($bm = 2$), for ten repetitions each, and measured the test set performance of their respective final incumbent parameter configurations. We report mean and standard deviation of performance across the ten repetitions of the configurator, and also report the test-set performance of the parameter configuration found in the repetition with the best *training-set* performance. Note that choosing the configuration found in the repetition with the best training-set performance is a perfectly legitimate procedure since it does not require knowledge of the test set. The only catch is that the running time grows by a factor of ten when we perform ten repetitions to keep the result of the best one.

8.3 Experimental Results

In Table 13, we compare performance of our algorithms' default parameter configurations with the final parameter configurations found by BasicILS(100) and FocusedILS for the nine BROAD configuration scenarios introduced in this section, as well as the five INDEPTH configuration scenarios we used throughout this paper (defined in Table 1 on page 9). Note that in some configuration scenarios (e.g., SPEAR-SWV, CPLEX-CLS, CPLEX-MIK) there was substantial variance between the different repetitions of the configurator, and the repetition with the best training performance yielded a

4. http://www.ieor.berkeley.edu/~atamturk/bcol/

Scenario	Default	Test performance (mean runtime over test instances, in CPU seconds)				Fig.
		mean ± stddev. for 10 repetitions		Repetition with best training performance		
		BasicILS	FocusedILS	BasicILS	FocusedILS	
Saps-SWGCP	20.41	0.32 ± 0.06	**0.32 ± 0.05**	**0.26**	**0.26**	14(a)
Saps-QCP	9.80	5.50 ± 0.53	**5.21 ± 0.39**	5.91	**5.20**	14(b)
Spear-SWGCP	9.74	**8.05 ± 0.9**	8.3 ± 1.1	6.8	**6.6**	14(c)
Spear-QCP	2.65	1.39 ± 0.33	**1.29 ± 0.2**	**1.16**	1.21	14(d)
Cplex-Regions100	1.61	0.5 ± 0.3	**0.35 ± 0.04**	0.35	**0.32**	14(e)
Spear-swv	424	95 ± 157	**10.4 ± 12.4**	4.21	**1.58**	15(a)
Spear-ibm	996	**996 ± 41**	1062 ± 170	1,030	**958**	15(b)
Saps-random	1,271	1,176 ± 43	**1,140 ± 30**	**1,180**	1,187	15(c)
Saps-crafted	1,556	**1,534 ± 64**	1,548 ± 53	1,492	**1,471**	15(d)
Cplex-Regions200	72	45 ± 24	**11.4 ± 0.9**	15	**10.5**	15(e)
Cplex-conic.sch	5.37	**2.27 ± 0.11**	2.4 ± 0.29	**2.14**	2.35	15(f)
Cplex-CLS	712	443 ± 294	**327 ± 860**	80	**23.4**	15(g)
Cplex-MIK	64.8	**20 ± 27**	301 ± 948 [5]	1.72	**1.19**	15(h)
Cplex-QP	969	**755 ± 214**	827 ± 306	528	**525**	15(i)

Table 13: Experimental results for our five InDepth and nine Broad configuration scenarios. For each configuration scenario, we list test performance (mean runtime over test instances) of the algorithm default, mean ± stddev of test performance across ten repetitions of BasicILS(100) & FocusedILS (run for two CPU days each), and the test performance of the repetition of BasicILS and FocusedILS that is best in terms of *training* performance. Boldface indicates the better of BasicILS and FocusedILS. The algorithm configurations found in FocusedILS's repetition with the best training performance are listed in an online appendix at `http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/results.html`. Column "Fig." gives a reference to a scatter plot comparing the performance of those configurations against the algorithm defaults.

parameter configuration that was also very good on the test set. We tended to observe higher variation for FocusedILS than for BasicILS. Even in configuration scenarios where BasicILS performed better on average, this high variance sometimes helped FocusedILS to achieve the repetition with the best training performance. Examples for this effect are the scenarios Spear-SWGCP, Spear-ibm, Saps-crafted, Cplex-MIK, and Cplex-QP.

While BasicILS outperformed FocusedILS in five of our nine Broad configuration scenarios in terms of mean test performance across ten repetitions, FocusedILS achieved the better test performance for the repetition with the best training performance for all but two Broad configuration scenarios (in which it performed almost as well). For some of the scenarios, FocusedILS performed much better than BasicILS, for example in Spear-swv, Cplex-Regions200, and Cplex-CLS.

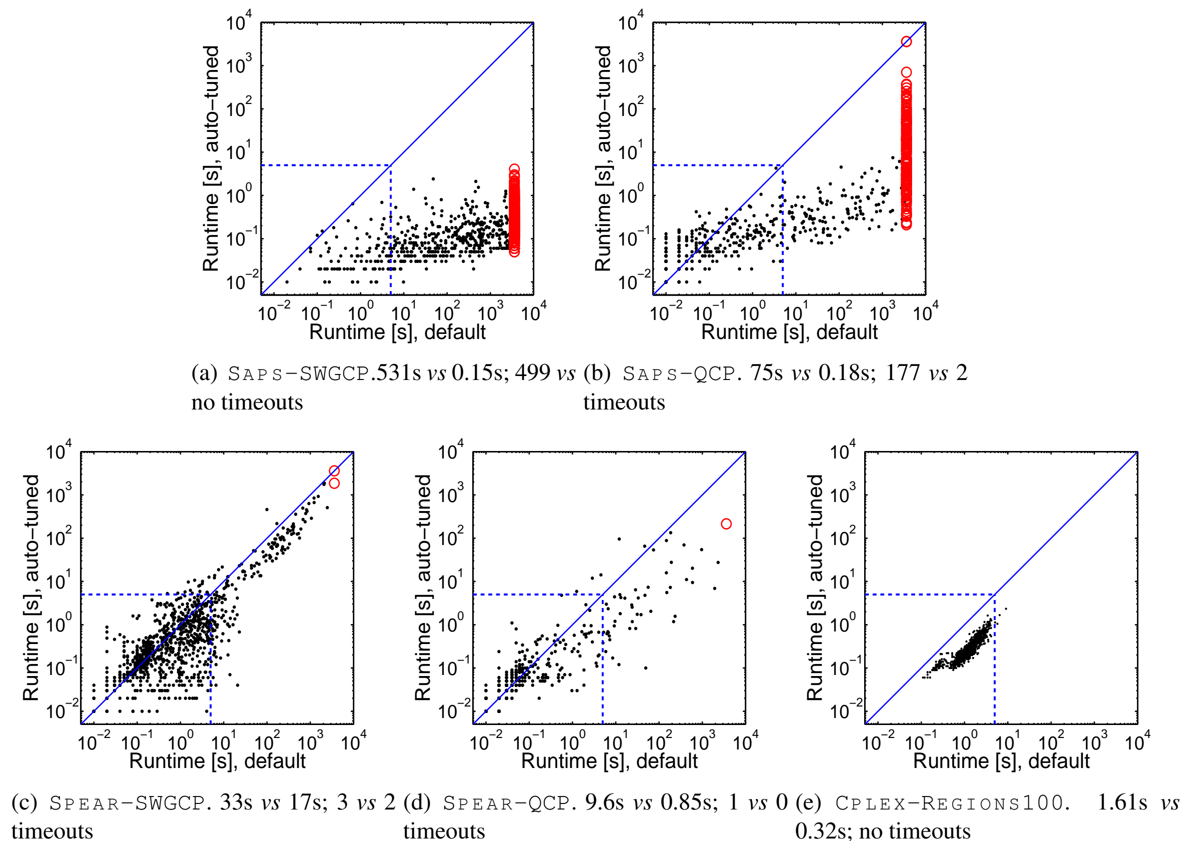
5. For configuration scenario Cplex-MIK, nine out of ten runs of FocusedILS yielded parameter configurations with average runtimes smaller than two seconds; one run, however, was very unfortunate. There, θ_{inc}, one of the first visited configurations solved the first instance π_1 in 0.84 seconds, and no other configuration managed to solve π_1 in less than $2 \times 0.84 = 1.68$ seconds; thus, every configuration $\theta \neq \theta_{inc}$ timed out on π_1 due to the aggressive capping strategy with $bm = 2$ used in these experiments. FocusedILS then iterated the following steps: perturbation to a new configuration θ; comparison of θ against a neighbour θ' using a single run each on π_1, both of which timed out after 1.68 seconds, breaking the tie in favour of θ' since $N(\theta) = N(\theta') = 1$; two bonus runs for $N(\theta')$; comparison of θ' against all its neighbours θ'' using a single run on π_1, and breaking ties in favour of θ' since $N(\theta') > N(\theta'')$; 202 bonus runs for θ'. In the seven iterations performed within two CPU days, this process did not find a configuration better than θ_{inc}, which did not manage to solve a single instance in the test set. Since the runtime of unsuccessful

Note that in all configuration scenarios we considered, both BasicILS and FocusedILS found parameter configurations that were better than the algorithm defaults, and that sometimes these new configurations were very substantially better. In Figure 14, we provide a scatter plot for each of our five INDEPTH configuration scenarios. These plots provide a comparison of the performance of the default parameter configuration and the configuration found in the repetition with best training performance. The speedups FocusedILS achieved over the default in terms of average runtimes ranged from factors of about two and ten for SPEAR-SWGCP and SPEAR-QCP to factors of over 400 and over 3,000 for SAPS-QCP and SAPS-SWGCP, respectively (see the subcaptions of Figure 14 for the exact numbers).

In Figure 15, we provide analogous scatter plots for each of our BROAD configuration scenarios. In some cases, results were excellent, e.g., for SPEAR-SWV: while the default timed out on 21 instances after a cutoff time of one CPU hour, the automatically configured version took no more than ten seconds to solve any instance. (In our earlier application paper (Hutter et al., 2007a) we chose larger cutoff times of 10 CPU hours for SPEAR-SWV and SPEAR-IBM and observed even more pronounced differences in mean runtime. Especially for SPEAR-IBM, where gains were larger for harder instances, the larger cutoff time led to a speedup factor of 4.5 as compared to 1.2 with the cutoff time we chose here.) For the SAPS scenarios SAPS-RANDOM and SAPS-CRAFTED, there was little correlation between the runtime with the default and the automatically determined parameter setting, though the latter configurations were somewhat better on average. For CPLEX-REGIONS200, CPLEX-CONIC.SCH, CPLEX-CLS, and CPLEX-MIK, speedups were quite consistent across instances (reaching from a speedup in terms of average runtime of a factor of 2 (CPLEX-CONIC.SCH) to an average factor of 20 (CPLEX-MIK). Finally, for CPLEX-QP, the optimized parameter configuration achieved good performance with the cutoff time used for the configuration process (300 CPU seconds, see Figure 16), but this performance did not carry over to the higher cutoff time we used in our tests (3600 CPU seconds, see Figure 15(i)).

We noticed several trends. Firstly, for homogeneous instance distributions the speedups tended to be higher than for heterogeneous instance collections. While SAPS performance improved greatly for the fairly homogeneous instance distributions QCP and SWGCP, we did not see a great improvement in the two configuration scenarios SAPS-RANDOM and SAPS-CRAFTED whose highly heterogeneous benchmark sets consisted of all random and crafted instances from five years of SAT competitions. Furthermore, in these heterogeneous scenarios, the runtimes for the default and the optimized parameter configuration were less correlated than in the other scenarios. This is intuitive since for heterogeneous distributions we cannot expect gains for a subset of instances to carry over to the entire distribution. Secondly, speedups over the default seemed to be similar for distributions with smaller and larger instances. In particular, for the very similar scenarios CPLEX-REGIONS100 and CPLEX-REGIONS200 we see slightly larger improvements for the configuration scenario with the harder instances (a factor of 6.8 *vs* a factor of 5.0). Finally, for some configuration scenarios (e.g., SPEAR-SWV, CPLEX-CLS, and CPLEX-MIK), speedups were much more pronounced for the hardest test instances. This is intuitive since we used mean runtime as an optimization objective, which naturally gives more weight to harder instances. Interestingly, this trend of larger speedups for harder instances also held for configuration scenario CPLEX-QP, up to the training cutoff time of 300 seconds (see Figure 16). However, if algorithm behaviour within the training cutoff time is

runs was counted as ten times the cutoff time, this resulted in an average runtime of $10 \times 300 = 3,000$ seconds for this unfortunate run. This demonstrates the risk of capping too aggressively, and underlines the importance of performing multiple runs of FocusedILS with different orderings of the training instances.



(a) SAPS-SWGCP.531s *vs* 0.15s; 499 *vs* no timeouts

(b) SAPS-QCP. 75s *vs* 0.18s; 177 *vs* 2 timeouts

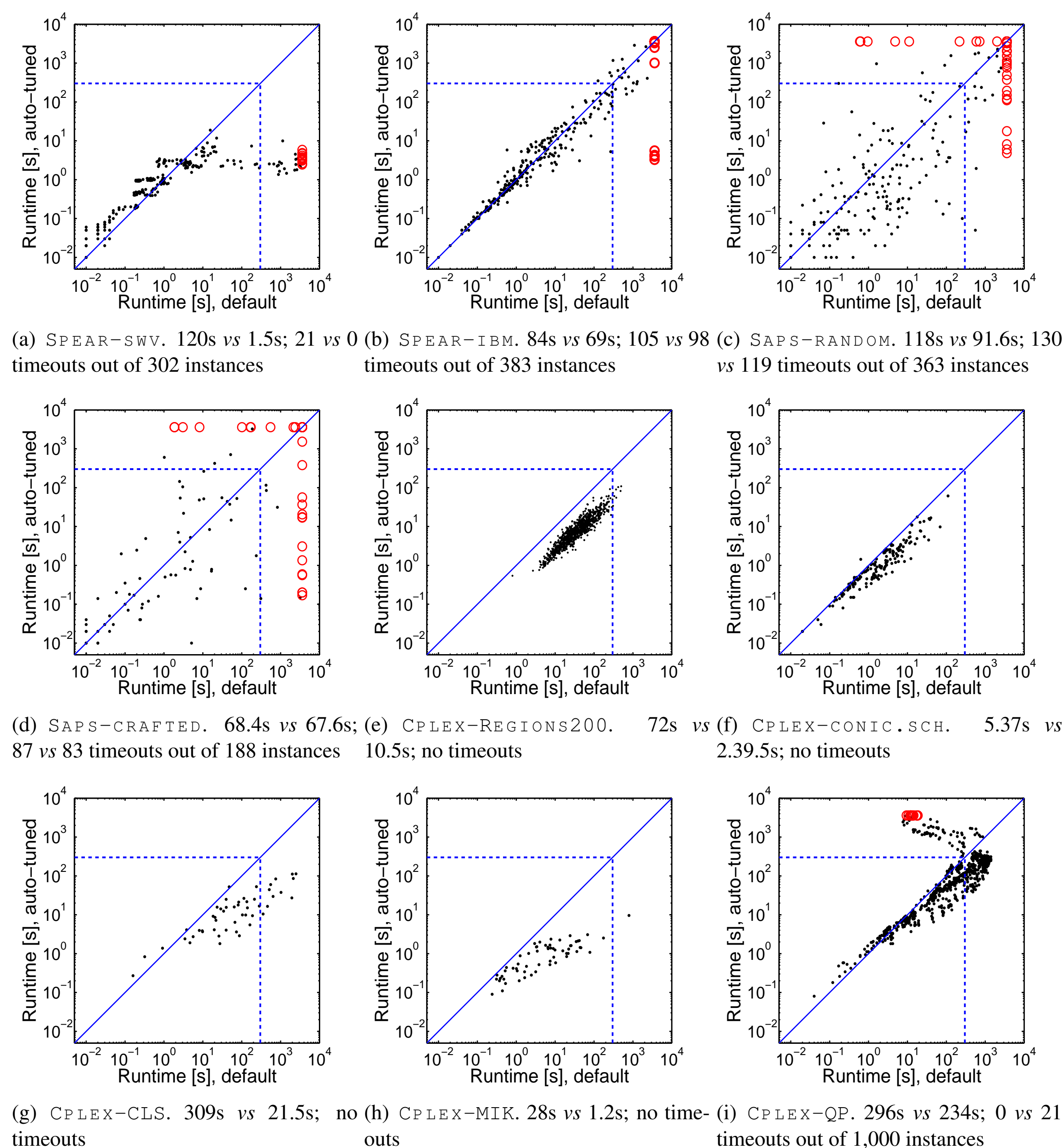
(c) SPEAR-SWGCP. 33s *vs* 17s; 3 *vs* 2 timeouts

(d) SPEAR-QCP. 9.6s *vs* 0.85s; 1 *vs* 0 timeouts

(e) CPLEX-REGIONS100. 1.61s *vs* 0.32s; no timeouts

Figure 14: Comparison of default *vs* automatically-determined parameter configurations for our five INDEPTH configuration scenarios. Each dot represents one test instance; time-outs (after one CPU hour) are denoted by red circles. The blue dashed line at five CPU seconds indicates the cutoff time of the target algorithm used during the configuration process. The subfigure captions give mean runtimes for the instances solved by each of the configurations (default *vs* optimized), as well as the number of timeouts for each. The automatically-determined configurations were obtained from the FocusedILS run with the best performance on the training set amongst 25 repetitions (each with a running time of up to five CPU hours). All data shown are based on test sets that are disjoint from the respective training sets.

not informative about behaviour in longer algorithm runs, performance with longer cutoffs may be poor. For example, see Figure 15(i); this underlines the importance of choosing the training cutoff time carefully.



(a) SPEAR-SWV. 120s *vs* 1.5s; 21 *vs* 0 timeouts out of 302 instances

(b) SPEAR-IBM. 84s *vs* 69s; 105 *vs* 98 timeouts out of 383 instances

(c) SAPS-RANDOM. 118s *vs* 91.6s; 130 *vs* 119 timeouts out of 363 instances

(d) SAPS-CRAFTED. 68.4s *vs* 67.6s; 87 *vs* 83 timeouts out of 188 instances

(e) CPLEX-REGIONS200. 72s *vs* 10.5s; no timeouts

(f) CPLEX-CONIC.SCH. 5.37s *vs* 2.39.5s; no timeouts

(g) CPLEX-CLS. 309s *vs* 21.5s; no timeouts

(h) CPLEX-MIK. 28s *vs* 1.2s; no timeouts

(i) CPLEX-QP. 296s *vs* 234s; 0 *vs* 21 timeouts out of 1,000 instances

Figure 15: Comparison of default *vs* automatically-determined parameter configuration for our nine BROAD configuration scenarios. Each dot represents one test instance; time-outs (after one CPU hour) are denoted by red circles. The blue dashed line at 300 CPU seconds indicates the cutoff time of the target algorithm used during the configuration process. The subfigure captions give mean runtimes for the instances solved by each of the configurations (default *vs* optimized), as well as the number of timeouts for each. The automatically-determined configurations were obtained from the FocusedILS run with the best performance on the training set amongst 10 repetitions (each with a running time of up to two CPU days). All data shown are based on test sets that are disjoint from the respective training sets.

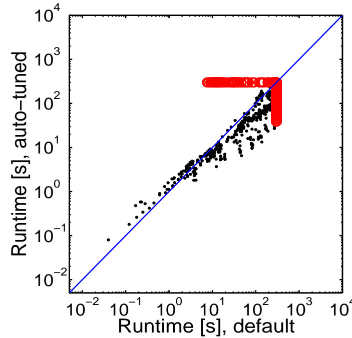


Figure 16: A plot identical to Figure 15(i), but with a test cutoff time of 300 CPU seconds (the same used for training) instead of 3600 CPU seconds. Although clutter obscures this fact in the figure, the default timed out on twice as many instances as the automatically-determined setting (305 *vs* 150 instances). For the instances solved under both parameter configurations, the average runtimes were 81 seconds (default) and 44 seconds (optimized). Finally, our scalar cost measure that counts timeouts as $10 \times$ cutoff time evaluated to 969 for the default and 525 for the optimized setting. Thus, the parameter configuration found by FocusedILS *did* generalize well to previously-seen test data, but *not* to larger runtimes.

9. ParamILS configuring itself

As a heuristic optimization procedure, ParamILS is itself controlled by a number of parameters: the number of random configurations, R, to be sampled at the beginning of search; the perturbation strength, s; and the probability of random restarts, p_{restart}. Furthermore, our aggressive capping mechanism makes use of an additional parameter: the bound multiplier, bm. In all experiments reported in previous sections, we have used the manually-determined default values $\langle R, s, p_{\text{restart}}, bm\rangle = \langle 10, 3, 0.01, 2\rangle$. However, since ParamILS is a general method for automated algorithm configuration, it is natural to wonder whether we could achieve improved performance by using ParamILS to configure its own parameters for a given configuration scenario.

Figure 17 illustrates the process of self-configuration. We use ParamILS with its default parameters as a *meta-configurator* in order to configure the *target configurator* ParamILS, which in turn is run and evaluated on instances of the algorithm configuration problem. These instances correspond to configuration scenarios, each of which consists of a parameterized *base algorithm* and a set of input data. In other words, configuration scenarios, such as SPEAR-QCP, play the same role in configuring the target configurator (here: ParamILS), as SAT instances do in configuring a SAT algorithm, such as SPEAR. The objective to be optimized is performance across a number of configuration scenarios.

For the self-configuration experiment described in the following, we chose FocusedILS with aggressive capping as the target configurator and used the sets of parameter values shown in Table 14. During the development of ParamILS, we had already considered making the perturbation strength s dependent on the number of parameters to be configured, but ended up not implementing this idea,

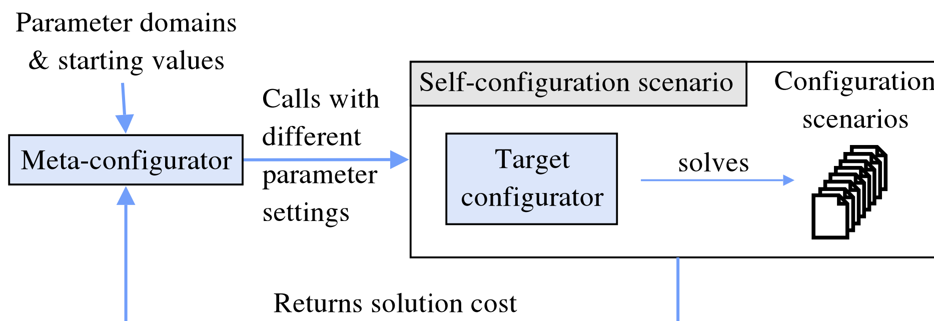


Figure 17: ParamILS configuring itself. The target algorithm is now ParamILS itself, and its "benchmark instances" are configuration scenarios. The meta-configurator ParamILS (with default parameters) searches for good parameter settings for the target configurator, ParamILS, evaluating each parameter setting by running ParamILS on the appropriate configuration scenarios, with geometric mean quality of the results serving as the performance measure. The same binary was used for the meta-configurator and the target configurator.

because we did not want to introduce too many parameters. However, equipped with an automated configuration tool, we now introduced a Boolean parameter that would switch between two conditional parameters: fixed absolute perturbation strength s_{abs} and relative perturbation strength s_{rel} (a factor to be multiplied by the number of parameters to be configured to yield different perturbation strengths in different configuration scenarios). In total, there were $4+4 = 8$ settings for perturbation strength, 2 different restart probabilities, 3 possible numbers of random steps in the beginning, and 5 options for the bound multiplier, leading to $8 \times 2 \times 3 \times 5 = 160$ possible parameter configurations for our target configurator, FocusedILS.

In order to run the self-configuration experiment within a reasonable amount of time on our cluster, parallelization turned out to be crucial. Because BasicILS is easier to parallelize than FocusedILS, we chose BasicILS(100) as the meta-configurator. Furthermore, to avoid potentially problematic feedback between them, we treated the meta-configurator and the target configurator as two distinct procedures with separate parameters; therefore, changes to the parameters of the target configurator, FocusedILS, had no impact on the parameters of the meta-configurator, BasicILS(100), which we kept fixed to the previously-discussed default values.

BasicILS(100) evaluated each parameter configuration θ of the target configurator, FocusedILS, by performing 20 runs of FocusedILS(θ) on each of our five INDEPTH configuration scenarios. Unlike in previous experiments, we limited the total running time of FocusedILS to one CPU hour per configuration scenario; this reduction was necessary to keep the overall computational burden of our experiments within reasonable limits.[6] The best parameter configuration of the base algorithm found in each of these one-hour runs of the target configurator was then evaluated on a validation set (consisting of the 1,000 training instances but using different random seeds), using 1,000 runs and

6. We note that, similarly to the effect observed for scenario CPLEX-QP in the previous section, there is a risk of finding a parameter configuration that only works well for runs up to one hour and poorly thereafter.

Parameter	Note	Values considered
r	Number of initial random configurations	0, ***10***, 100
bm	Bound multiplier for adaptive capping	1.1, 1.3, 1.5, *2*, **4**
$p_{restart}$	Probability of random restart	*0.01*, **0.05**
pert	Type of perturbation	*absolute*, **relative**
s_{abs}	Absolute perturbation strength; only active when *pert*=absolute	1,*3*,5,10
s_{rel}	Relative perturbation strength; only active when *pert*=relative	0.1,0.2,**0.4**,0.8

Table 14: Parameters of ParamILS, the possible values considered for the self-configuration, and the values determined for FocusedILS in the self-configuration experiment. Values in italic font are the default values, bold faced values are chosen by the meta-configurator. (Default: $\langle r, bm, p_{restart}, pert, s_{\text{abs}}\rangle = \langle 10, 2, 0.01, \text{absolute}, 3\rangle$; Self-configured: $\langle r, bm, p_{restart}, pert, s_{\text{rel}}\rangle = \langle 10, 4, 0.05, \text{relative}, 0.4\rangle$) In the case of relative perturbation strength, we computed the actual perturbation strength as $\max(2, s_{\text{rel}} \times M)$, where M is the number of parameters to be set in a configuration scenario.

a cutoff time of five CPU seconds. Thus, including validation, each run of the target configurator required up to approximately 2.5 CPU hours (one hour for the configuration process[7] plus up to 5000 CPU seconds for validation).

The meta-configurator used the geometric mean of the 100 validation results obtained in this manner from the 20 runs for each of the 5 configuration scenarios to assess the performance of configuration θ. (We used geometric instead of arithmetic means because the latter can easily be dominated by the results from a single configuration scenario.) We note that, while this performance measure is ultimately based on algorithm runtimes (namely those of the base algorithms configured by the target configurator), unlike the objective used in previous sections, it does not correspond to the runtime of the target configurator itself (here: ParamILS). Rather, it corresponds to the solution quality the target configurator achieves within bounded runtime (one hour).

The 100 runs of the target configurator performed in order to evaluate each parameter configuration were run concurrently on 100 CPUs.We ran the meta-configurator BasicILS(100) for a total of five iterations in which 81 parameter configurations were evaluated. This process took a total of about 500 CPU days (where some of the 100 CPUs used in the experiment were occasionally idle after validation tasks requiring less than 5000 CPU seconds). The best configuration of the target configurator, FocusedILS, found in this experiment is shown in Table 14; it had a geometric mean runtime of 1.40 CPU seconds across the 100 validation runs (as compared to a geometric mean runtime of 1.45 achieved using FocusedILS's default settings) and was found after a total runtime of 71 CPU days; it was the 17th of the 81 configurations evaluated by BasicILS(100) and corresponded

7. ParamILS's CPU time is typically dominated by the time spent running the target algorithm. However, in configuration scenarios where each execution of the base algorithm is very fast this was not always the case, because of overhead due to bookkeeping and calling algorithms on the command line. In order to achieve a wall clock time close to one hour for each ParamILS run in the self-configuration experiments, we minimized the amount of bookkeeping in our simple Ruby implementation of ParamILS and counted every algorithm execution as taking at least one second. This lower bound on algorithm runtimes is a parameter of ParamILS that was set to 0.1 seconds for all other experimental results.

Scenario	Default ParamILS	Self-configured ParamILS	p-value
SAPS-SWGCP	$\mathbf{0.316 \pm 0.054}$	0.321 ± 0.043	(0.50)
SPEAR-SWGCP	8.30 ± 1.06	$\mathbf{8.26 \pm 0.88}$	0.72
SAPS-QCP	5.21 ± 0.39	$\mathbf{5.16 \pm 0.32}$	0.69
SPEAR-QCP	1.29 ± 0.20	$\mathbf{1.22 \pm 0.18}$	0.28
CPLEX-REGIONS100	0.35 ± 0.05	$\mathbf{0.34 \pm 0.02}$	0.34

Table 15: Comparison of test set performance (mean runtime of best configuration found, in CPU seconds) for FocusedILS with its default parameter settings, and with the parameter settings found via self-configuration. For each configuration scenario, we report mean $\pm$ stddev of the test performance over 25 repetitions of the configurators, and the p-value for a paired Max-Wilcoxon test (see Section 3.1.3).

to the first local minimum reached. The next three iterations of BasicILS all led to different and slightly worse configurations. In the fifth iteration, the same best configuration was found again.

Table 15 reports the performance achieved by FocusedILS with this new, automatically determined parameter settings, on the original INDEPTH configuration scenarios. We note that the performance measure used here differs from the objective optimized by the meta-configurator, BasicILS (namely, geometric mean runtimes over all scenarios with reduced running times). Nevertheless, the self-configuration process resulted in minor performance improvements in four out of five configuration scenarios. Upon closer examination, these performance differences turned out not to be statistically significant. Thus, while ParamILS appears to be capable of configuring itself, this computationally expensive process led to merely marginal improvements. We believe that this reflects the fact that the performance of FocusedILS (at least for the INDEPTH configuration scenarios considered here) is much more robust with respect to parameter settings than that of many other, highly-parameterised heuristic algorithms, and that, based on our insights into and experience with the ParamILS framework, we happened to previously choose close-to-optimal default parameter values.

10. Related Work

Many researchers before us have been dissatisfied with manual algorithm configuration, and various fields have developed their own approaches for automatic parameter tuning. We start this section with the most closely-related work—approaches that employ direct search to find good parameter configurations—and then describe other methods. Finally, we discuss work on related problems, such as finding the best parameter configuration or algorithm on a per-instance basis, and approaches that adapt their parameters during an algorithm's execution.

10.1 Direct Search Methods for Algorithm Configuration

Approaches for automated algorithm configuration go back to the early nineties, when a number of systems were developed for *adaptive problem solving*. One of these systems is Composer (Gratch and Dejong, 1992), which performs a hill-climbing search in configuration space, taking moves if enough evidence has been gathered to render a neighbouring configuration statistically significantly

better than the current configuration. Composer was successfully applied to improving the five parameters of an algorithm for scheduling communication between a collection of ground-based antennas and aircraft in deep space (Gratch and Chien, 1996).

Around the same time, the MULTI-TAC system was introduced by Minton (1993, 1996). MULTI-TAC takes as input a number of generic heuristics as well as a specific problem domain and a distribution over problem instances. It adapts the generic heuristics to the problem domain and automatically generates domain-specific LISP programs implementing them. A beam search is then used to choose the best LISP program where each program is evaluated by running it on a number of problem instances sampled from the given distribution.

Another search-based approach that uses a fixed training set was introduced by Coy et al. (2001). Their approach works in two steps. First it finds a good parameter configuration θ_i for each instance I_i in the training set by a combination of experimental design (full factorial or fractional factorial) and gradient descent. Then it combines the parameter configurations $\theta_1, \ldots, \theta_N$ thus determined, by setting each parameter to the average of the values taken in all of them. Observe that this averaging step restricts the applicability of the method to algorithms with exclusively numerical parameters.

A similar approach, also based on a combination of experimental design and gradient descent, using a fixed training set for evaluation, is implemented in the CALIBRA system of Adenso-Diaz and Laguna (2006). CALIBRA starts by evaluating each parameter configuration in a full factorial design with two values per parameter. It then iteratively homes in to good regions of parameter configuration space by employing fractional experimental designs that evaluate nine configurations around the best performing configuration found so far. The grid for the experimental design is refined in each iteration, which provides a nice solution to the automatic discretization of continuous parameters. Once a local optimum is found, the search is restarted (with a coarser grid) by combining some of the best configurations found so far, but also introducing some noise to encourage diversification. The experiments reported by Adenso-Diaz and Laguna (2006) show great promise in that CALIBRA was able to find parameter settings for six independent algorithms that matched or outperformed the respective originally proposed parameter configurations. CALIBRA's main drawback is its limitation to tuning numerical and ordinal parameters, and to tuning a maximum of five free parameters.

When we first introduced ParamILS, we ran experiments with CALIBRA, BasicILS, and FocusedILS for four configuration scenarios (Hutter et al., 2007b). In two scenarios, our algorithms performed better than CALIBRA, on the third scenario there was no statistically significant difference, and for the fourth CALIBRA was not applicable because the algorithm had too many parameters and some of them were categorical. (This current article focuses on complex algorithms with large numbers of parameters, and we thus do not repeat the comparison with CALIBRA.)

Oltean (2005) employed linear genetic programming to evolve genetic algorithms. This approach has its roots in automatically finding computer programs to solve a given problem, and was adapted to search for the best combination of genetic programming operators, such as mutation, crossover, and selection of individuals. His experiments show that the automatically-evolved genetic algorithms outperformed standard implementations of genetic algorithms on a variety of tasks, such as function optimization, the travelling salesperson and the quadratic assignment problem (Oltean, 2005). However, as most of the approaches we surveyed so far (with the notable exception of Composer), this work did not use a separate test set to evaluate the performance of the final configurations found; thus, it is unclear to what extent the reported results reflect overconfidence of the configurator.

There is also work on automated parameter tuning in the numerical optimization literature. In particular, Audet and Orban (2006) propose the mesh adaptive direct search algorithm. Designed for purely continuous parameter configuration spaces, this algorithm is guaranteed to converge to a *local* optimum of the cost function. Parameter configurations were evaluated on a fixed set of large unconstrained regular problems from the CUTEr collection, using runtime and number of function calls necessary for solving the problem as optimization objectives. Performance improvements of around 25% over the classical configuration of four continuous parameters of interior point methods were reported.

Algorithm configuration is a stochastic optimization problem, and there exists a large body of algorithms designed for such problems (see, e.g., Spall, 2003). However, many of the algorithms in the stochastic optimization literature require explicit gradient information and are thus inapplicable to algorithm configuration. There exist a number of algorithms that approximate the gradient from function evaluations only (e.g., by finite differences), and provably converge to a local minimum of the cost function under mild conditions, such as continuity. Still, these methods are primarily designed to deal with numerical parameters and only find local minima. We are not aware of any applications of general purpose algorithms for stochastic optimization to algorithm configuration.

10.2 Other Methods for Algorithm Configuration

Sequential parameter optimization (SPO) (Bartz-Beielstein, 2006) is a model-based parameter optimization approach. It starts by running the target algorithm with parameter configurations from a Latin hypercube design on a number of training instances. It then builds a response surface model based on Gaussian process regression and uses the model's predictions and predictive uncertainties to determine the next parameter configuration to evaluate. The metric underlying the choice of promising parameter configurations is the expected improvement criterion (Jones et al., 1998). After each algorithm run the response surface is refitted, and a new parameter configuration is determined based on the updated model. In contrast to the previously-mentioned methods, SPO does not use a fixed training set; instead, it starts with a small training set and doubles its size every time a parameter configuration is determined to be optimal for two iterations in a row. The main drawback of SPO is its limitation to continuous parameters.

Another approach is based on adaptations of racing algorithms in machine learning to the algorithm configuration problem. Birattari et al. (2002; 2005) applied F-races to the configuration of stochastic local search algorithms. Their procedure takes as input an algorithm $\mathcal{A}$, a finite set of algorithm configurations Θ, and an instance distribution $\mathcal{D}$. It iteratively runs the target algorithm with all "surviving" parameter configurations on a number of instances sampled from $\mathcal{D}$ (in the simplest case, each iteration runs all surviving configurations on one instance). A configuration is eliminated from the race as soon as enough statistical evidence is gathered against it: after each iteration first the non-parametric Friedman test is applied to check whether there are significant differences among the configurations. If this is the case, the inferior-performing configurations are eliminated, based on the results of Friedman post-tests. This process is iterated until only one configuration survives or a given cutoff time is reached. Various applications of F-race have demonstrated very good performance (for an overview, see Birattari, 2005). However, since at the start of the procedure all candidate configurations are evaluated, this approach is limited to situations in which the number of candidate configurations considered simultaneously is not too large. In fact, published experiments with F-race have been limited to applications with only around 1200 config-

urations. A recent extension presented by Balaprakash et al. (2007) iteratively performs F-races on differently-defined subsets of parameter configurations. This approach scales better to large configuration spaces, but the version discussed in (Balaprakash et al., 2007) was described for algorithms with numerical parameters only.

10.3 Related Algorithm Configuration Problems

So far in this section—and indeed, throughout this article—we have focused on the problem of finding the best algorithm configuration for an entire set (or distribution) of problem instances. Related approaches attempt to find the best configuration or algorithm on a per-instance basis, or to adapt algorithm parameters during the execution of an algorithm. Approaches for setting parameters on a per-instance basis have been, for example, introduced in (Patterson and Kautz, 2001; Cavazos and O'Boyle, 2006; Agakov et al., 2006; Hutter et al., 2006). For approaches that try to choose the best *algorithm* on a per-instance basis, see (Leyton-Brown et al., 2002; Carchrae and Beck, 2005; Gebruers et al., 2005; Gagliolo and Schmidhuber, 2006; Xu et al., 2008). Other related work makes online decisions about when to restart an algorithm (Horvitz et al., 2001; Kautz et al., 2002; Gagliolo and Schmidhuber, 2007). Finally, so-called reactive search methods modify algorithm parameters during the algorithm trajectory (Battiti et al., 2008). This strategy can be seen as complementary to our work: even reactive search methods tend to have a number of parameters that remain fixed during the search and can hence be configured by offline approaches.

11. Conclusions and Future work

In this work, we studied the problem of automatically configuring the parameters of complex, heuristic algorithms in order to optimise performance on a given set of benchmark instances. Building on our previous work, we gained further insights into this commonly encountered and practically relevant algorithm configuration problem (such as, for example, the importance of the cutoff time used during configuration). We also extended our earlier algorithm configuration procedure, ParamILS, with a new capping mechanism and obtained excellent results when applying the resulting enhanced version of ParamILS to two high-performance SAT algorithms as well as to CPLEX and a wide range of benchmark sets.

On test sets disjoint from the training set available to ParamILS, the parameter configurations ParamILS found almost always outperformed the algorithms' carefully-chosen default configurations, for some configuration scenarios by as much as two orders of magnitude. The improvements over CPLEX's default parameter configuration are particularly noteworthy. To the best of our knowledge, ours is the first published work on automatically configuring this algorithm. We do *not* claim to have found a new parameter configuration for CPLEX that is uniformly better than its default. Rather, given a somewhat homogeneous instance set, we find a configuration specific to that set that typically outperforms the default, sometimes by a factor as high as 20. Note that we achieved these results even though we are *not* intimately familiar with CPLEX and its parameters; we chose the parameters to optimize as well as the values to consider based on a single person-day of studying the CPLEX user manual. The success of automated algorithm configuration even under these extreme conditions demonstrates the potential of the approach.

The ParamILS source code and executable are freely available at

`http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/`,

along with a quickstart guide and benchmark data for the configuration scenarios studied in this article. Briefly, in order to employ ParamILS, a user must discretize the target algorithm's parameter configuration space and choose a representative instance set of interest as well as a cutoff time for each algorithm run. To safeguard against over-tuning, the instance set should be large enough and split into disjoint training and test sets whenever possible. The cutoff time for each run should be chosen such that target algorithm behaviour within that time is representative of behaviour at true runtimes of interest.

Overall, we firmly believe that automated algorithm configuration methods such as ParamILS will play an increasingly prominent role in the development of high-performance algorithms and their applications. The study of such methods is a rich and fruitful research area with many interesting questions remaining to be explored.

In ongoing work, we are currently developing methods that adaptively adjust the domains of integer-valued and continuous parameters during the configuration process. Similarly, we plan to enhance ParamILS with dedicated methods for dealing with continuous parameters that do not require discretisation by the user.

Another direction for further development concerns the strategic selection of problem instances used during evaluation of configurations and of instance-specific cutoff times used in this context. By heuristically preventing the configuration procedure from spending inordinate amounts of time in trying to evaluate poor parameter settings on very hard problem instances, it should be possible to improve its scalability.

We believe that there is significant room for combining aspects of the methods studied here with concepts from related work on this and similar algorithm configuration problems. In particular, we believe it would be fruitful to integrate statistical testing methods — as used, e.g., in F-race — into ParamILS. Furthermore, we see much potential in the use of response surface models and active learning, and believe these can be combined with our approach.

Finally, while the algorithm configuration problem studied in this article is of significant practical importance, there is also much to be gained from studying methods for related problems, in particular, instance-specific algorithm configuration and on-line adjustment of parameters during the run of an algorithm.

Acknowledgements

We would like to thank Kevin Murphy for many helpful discussions regarding this work. We would also like to thank Domagoj Babić, the author of SPEAR, and Dave Tompkins, the author of the UBCSAT SAPS implementation we used in our experiments. Finally, we would like to thank the researchers who provided the instances or instance generators used in our work, in particular Gent et al. (1999), Gomes and Selman (1997), Leyton-Brown et al. (2000), Babić and Hu (2007), Zarpas (2005), Le Berre and Simon (2004), Aktürk et al. (2007), Atamtürk and Muñoz (2004), Atamtürk (2003), and Andronescu et al. (2007). Using the aforementioned generators, Lin Xu created the specific sets of QCP and SW-GCP instances we used. Thanks also to Chris Fawcett and Ashique KhudaBukhsh for their comments on a draft of this article.

References

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

Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F. P., Thomson, J., Toussaint, M., and Williams, C. K. I. (2006). Using machine learning to focus iterative optimization. In *Proceedings of the Fourth Annual International Symposium on Code Generation and Optimization (CGO-06)*, pages 295–305, Washington, DC, USA. IEEE Computer Society.

Aktürk, S. M., Atamtürk, A., and Gürel, S. (2007). A strong conic quadratic reformulation for machine-job assignment with controllable processing times. Research Report BCOL.07.01, University of California-Berkeley.

Andronescu, M., Condon, A., Hoos, H. H., Mathews, D. H., and Murphy, K. P. (2007). Efficient parameter estimation for RNA secondary structure prediction. *Bioinformatics*, 23:i19–i28.

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

Atamtürk, A. and Muñoz, J. C. (2004). A study of the lot-sizing polytope. *Mathematical Programming*, 99:443–465.

Audet, C. and Orban, D. (2006). Finding optimal algorithmic parameters using the mesh adaptive direct search algorithm. *SIAM Journal on Optimization*, 17(3):642–664.

Babić, D. and Hu, A. J. (2007). Structural Abstraction of Software Verification Conditions. In W. Damm, H. H., editor, *Computer Aided Verification: 19th International Conference, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 366–378. Springer Verlag, Berlin, Germany.

Balaprakash, P., Birattari, M., and Stützle, T. (2007). Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement. In Bartz-Beielstein, T., Aguilera, M. J. B., Blum, C., Naujoks, B., Roli, A., Rudolph, G., and Sampels, M., editors, *4th International Workshop on Hybrid Metaheuristics (MH'07)*, pages 108–122.

Bartz-Beielstein, T. (2006). *Experimental Research in Evolutionary Computation: The New Experimentalism*. Natural Computing Series. Springer Verlag, Berlin, Germany.

Battiti, R., Brunato, M., and Mascia, F. (2008). *Reactive Search and Intelligent Optimization*, volume 45 of *Operations research/Computer Science Interfaces*. Springer Verlag. In press; available online at http://reactive-search.org/thebook.

Birattari, M. (2004). *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*. PhD thesis, Université Libre de Bruxelles, Brussels, Belgium. Available online at http://iridia.ulb.ac.be/˜mbiro/thesis/.

Birattari, M. (2005). *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*. DISKI 292, Infix/Aka, Berlin, Germany.

Birattari, M., Stützle, T., Paquete, L., and Varrentrapp, K. (2002). A racing algorithm for configuring metaheuristics. In Langdon, W. B., Cantu-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M. A., Schultz, A. C., Miller, J. F., Burke, E., and Jonoska, N., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pages 11–18. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Carchrae, T. and Beck, J. C. (2005). Applying machine learning to low-knowledge control of optimization algorithms. *Computational Intelligence*, 21(4):372–387.

Cavazos, J. and O'Boyle, M. F. P. (2006). Method-specific dynamic compilation using logistic regression. In Cook, W. R., editor, *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-06)*, pages 229–240, New York, NY, USA. ACM Press.

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

den Besten, M. L., Stützle, T., and Dorigo, M. (2001). Design of iterated local search algorithms: An example application to the single machine total weighted tardiness problem. In *Applications of Evolutionary Computing*, volume 2037 of *Lecture Notes in Computer Science*, pages 441–452. Springer Verlag, Berlin, Germany.

Diao, Y., Eskesen, F., Froehlich, S., Hellerstein, J. L., Spainhower, L., and Surendra, M. (2003). Generic online optimization of multiple configuration parameters with application to a database server. In Brunner, M. and Keller, A., editors, *14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM-03)*, volume 2867 of *Lecture Notes in Computer Science*, pages 3–15. Springer Verlag, Berlin, Germany.

Gagliolo, M. and Schmidhuber, J. (2006). Dynamic algorithm portfolios. In Amato, C., Bernstein, D., and Zilberstein, S., editors, *Ninth International Symposium on Artificial Intelligence and Mathematics (AI-MATH-06)*.

Gagliolo, M. and Schmidhuber, J. (2007). Learning restart strategies. In Veloso, M. M., editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, volume 1, pages 792–797. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Gebruers, C., Hnich, B., Bridge, D., and Freuder, E. (2005). Using CBR to select solution strategies in constraint programming. In Muñoz-Avila, H. and Ricci, F., editors, *Proceedings of the 6th International Conference on Case Based Reasoning (ICCBR'05)*, volume 3620 of *Lecture Notes in Computer Science*, pages 222–236. Springer Verlag, Berlin, Germany.

Gent, I. P., Hoos, H. H., Prosser, P., and Walsh, T. (1999). Morphing: Combining structure and randomness. In Hendler, J. and Subramanian, D., editors, *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, pages 654–660, Orlando, Florida. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Gomes, C. P. and Selman, B. (1997). Problem structure in the presence of perturbations. In Kuipers, B. and Webber, B., editors, *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 221–226. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Gratch, J. and Chien, S. A. (1996). Adaptive problem-solving for large-scale scheduling problems: A case study. *Journal of Artificial Intelligence Research*, 4:365–396.

Gratch, J. and Dejong, G. (1992). Composer: A probabilistic solution to the utility problem in speed-up learning. In Rosenbloom, P. and Szolovits, P., editors, *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 235–240. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Hastie, T., Tibshirani, R., and Friedman, J. H. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. Springer Verlag.

Horvitz, E., Ruan, Y., Gomes, C. P., Kautz, H., Selman, B., and Chickering, D. M. (2001). A Bayesian approach to tackling hard computational problems. In Breese, J. S. and Koller, D., editors, *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence (UAI'01)*, pages 235–244. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Hutter, F., Babić, D., Hoos, H. H., and Hu, A. J. (2007a). Boosting Verification by Automatic Tuning of Decision Procedures. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*, pages 27–34, Washington, DC, USA. IEEE Computer Society.

Hutter, F., Hamadi, Y., Hoos, H. H., and Leyton-Brown, K. (2006). Performance prediction and automated tuning of randomized and parametric algorithms. In Benhamou, F., editor, *Principles and Practice of Constraint Programming – CP 2006: Twelfth International Conference*, volume 4204 of *Lecture Notes in Computer Science*, pages 213–228. Springer Verlag, Berlin, Germany.

Hutter, F., Hoos, H. H., and Stützle, T. (2007b). Automatic algorithm configuration based on local search. In Howe, A. and Holte, R. C., editors, *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI'07)*, pages 1152–1157. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Hutter, F., Tompkins, D. A. D., and Hoos, H. H. (2002). Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In Hentenryck, P. V., editor, *Principles and Practice of Constraint Programming – CP 2002: Eighth International Conference*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248. Springer Verlag, Berlin, Germany.

Johnson, D. S. (2002). A theoretician's guide to the experimental analysis of algorithms. In Goldwasser, M. H., Johnson, D. S., and McGeoch, C. C., editors, *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, pages 215–250. American Mathematical Society, Providence, RI, USA.

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

Kautz, H., Horvitz, E., Ruan, Y., Gomes, C. P., and Selman, B. (2002). Dynamic restart policies. In Dechter, R., Kearns, M., and Sutton, R., editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'02)*, pages 674–681. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Le Berre, D. and Simon, L. (2004). Fifty-five solvers in Vancouver: The SAT 2004 competition. In Hoos, H. H. and Mitchell, D. G., editors, *Theory and Applications of Satisfiability Testing: Proceedings of the Seventh International Conference (SAT'04)*, volume 3542 of *Lecture Notes in Computer Science*, pages 321–344. Springer Verlag.

Leyton-Brown, K., Nudelman, E., and Shoham, Y. (2002). Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In Hentenryck, P. V., editor, *Principles and Practice of Constraint Programming – CP 2002: Eighth International Conference*, volume 2470 of *Lecture Notes in Computer Science*, pages 556–572. Springer Verlag, Berlin, Germany.

Leyton-Brown, K., Pearson, M., and Shoham, Y. (2000). Towards a universal test suite for combinatorial auction algorithms. In Jhingran, A., Mason, J. M., and Tygar, D., editors, *EC '00: Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76, New York, NY, USA. ACM.

Lourenço, H. R., Martin, O., and Stützle, T. (2002). Iterated local search. In Glover, F. and Kochenberger, G., editors, *Handbook of Metaheuristics*, pages 321–353. Kluwer Academic Publishers, Norwell, MA, USA.

Minton, S. (1993). An analytic learning system for specializing heuristics. In Bajcsy, R., editor, *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 922–929. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Minton, S. (1996). Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1):1–40.

Oltean, M. (2005). Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410.

Patterson, D. J. and Kautz, H. (2001). Auto-WalkSAT: a self-tuning implementation of WalkSAT. In *Electronic Notes in Discrete Mathematics (ENDM), 9*.

Ridge, E. and Kudenko, D. (2006). Sequential experiment designs for screening and tuning parameters of stochastic heuristics. In Paquete, L., Chiarandini, M., and Basso, D., editors, *Workshop on Empirical Methods for the Analysis of Algorithms at the Ninth International Conference on Parallel Problem Solving from Nature (PPSN)*, pages 27–34.

Spall, J. C. (2003). *Introduction to Stochastic Search and Optimization*. John Wiley & Sons, Inc., New York, NY, USA.

Tompkins, D. A. D. and Hoos, H. H. (2004). UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT & MAX-SAT. In *Theory and Applications of Satisfiability Testing: Proceedings of the Seventh International Conference (SAT'04)*, volume 3542, pages 306–320. Springer Verlag, Berlin, Germany.

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

Zarpas, E. (2005). Benchmarking SAT Solvers for Bounded Model Checking. In Bacchus, F. and Walsh, T., editors, *Theory and Applications of Satisfiability Testing: Proceedings of the Eighth International Conference (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 340–354. Springer Verlag.