Using Racing to Automatically Configure Algorithms for Scaling Performance

James Styles and Holger Hoos

University of British Columbia, Canada, email: [jastyles,hoos]@cs.ubc.ca

Abstract. Automated algorithm configuration has been proven to be an effective approach for achieving improved performance of solvers for many computationally hard problems. Following our previous work, we consider the challenging situation where the kind of problem instances for which we desire optimised performance is too difficult to be used during the configuration process. In this work, we propose a novel combination of racing techniques with existing algorithm configurators to meet this challenge. We demonstrate that the resulting algorithm configuration protocol achieves better results than previous approaches and in many cases closely matches the bound on performance obtained using an oracle selector.

1 Introduction

High performance algorithms for computationally hard problems often have numerous parameters which control their behaviour and performance. Finding good values for these parameters, some exposed to end users and others hidden as hard-coded design choices and magic constants, can be a challenging problem for algorithm designers. Recent work on automatically configuring algorithms has proven to be very effective [4, 9, 10, 13, 14, 15, 18, 19]. These automatic algorithm configurators rely on the use of significant computational resource (e.g., large computer clusters) to explore the design space of an algorithm by evaluating thousands of different configurations. The general protocol for using an automatic configurator is as follows:

1. Identify the intended use case of the algorithm (e.g., structure and size of expected problem instances, resource limitations) and define a metric to be optimized (e.g., runtime).
2. Construct a training set which is representative of the intended use case. The performance of the configurator depends on being able to evaluate a large number, ideally thousands, of configurations. Training instances must be chosen to permit this.
3. Perform multiple independent runs of the configurator.
4. Validate the final configurations found by each configurator run and select the best performing configuration.

The choice of training set and validation method both have a significant impact on the quality of configuration found. The standard version of this protocol constructs a training set by drawing instances directly from the intended use

case and validates the configurations by evaluating them on the entire training set. The validation step in the standard version of the protocol is only needed to ensure configurations can be fairly compared, since in practice, a run of a configurator typically only uses a subset of the training set.

However, there are scenarios where instances in the intended use case are too difficult (i.e. require too much time to be solved) to be feasibly used in a training set. We consider an instance as being too difficult to use during training if it does not permit at least a thousand evaluations during the run of a configuator; notice that this definition of difficulty is relative to the time budget available for configuration. We have previously shown that the standard configuration protocol can be very ineffective for these scenarios and often produces configurations that perform worse than the default [17]. We also introduced a modification to the standard protocol based on the idea of using so-called intermediate instances, described in Section 2, to validate configurations. This new protocol was shown to reliably outperform the standard protocol for three very different domains (TSP, MIP and computer Go).

In this work, we show how even better configurations can be found using two novel configuration protocols that combine the idea of using intermediate problem instances for validation with the concept of racing. One of these protocols uses a new variant of F-Race [8] and the other is based on a novel racing procedure dubbed *ordered permutation race*. We show that both racing-based protocols reliably outperform our previous protocols [17] and are able to produce configurations up to 25% better within the same overall time budget or, equivalently, configurations of the same quality in up to 45% less total time and up to 90% less time for validation.

To assess the effectiveness of our racing-based protocols, we performed a large empirical study across five configuration scenarios. The first scenario involves Keld Helsgaun's implementation of the Lin-Kerninghan algorithm (LKH) [12], the state-of-the-art incomplete solver for the TSP; this scenario is known to be challenging for the standard protocol [17]. The second scenario considers configuring Lingeling [2], a state-of-the-art complete solver for the boolean satisfiability problem (SAT), for solving industrial instances from SAT Competition and SAT Race [3]. The remaining three scenarios consider configuring the state-of-the-art industrial mixed integer programming (MIP) solver CPLEX for instances modeled on real world data. We consider two scenarios based on configuring CPLEX [1] for solving CORLAT (see Section 4): one using CPLEX 12.1, where the standard protocol is known to be effective [13], and another using CPLEX 12.3. The final scenario is based configuring CPLEX 12.3 for solving a class of application instances that can take up to 20 hours each to solve (see Section 4). All scenarios use the freely available and highly effective automated algorithm configurators ParamILS [15] and SMAC [14].

The remainder of this paper is structured as follows. Section 2 summarises the train-easy select-intermediate (TE-SI) protocol that provided the starting point for this work. Section 3 introduces our new, racing-based protocols. Section 4 describes the five configuration scenarios and the experimental setup we used to assess the various configuration protocols, and Section 5 describes the results

from our computational experiments. Finally, Section 6 provides conclusions and an overview of ongoing and future work.

2 Validation using Intermediate Instances

The performance of automated algorithm configurators depends heavily on the number of evaluations that can be performed within a given time budget and the guidance that each evaluation provides. In scenarios where the intended use case requires very difficult problem instances to be solved, the number of evaluations performed when following the standard protocol can become too small to be effective (in extreme cases, it may be impossible to perform even a single evaluation). There are three ways to address this situation:

1. Continue to use the hard instances and just accept the reduced performance from performing fewer evaluations. This option can be successful if there exists easy to find configurations which are significantly better than the default.
2. Continue to use the hard instances and enforce strict per-instance time cutoffs to guarantee a minimum number of evaluations. This option requires that configurations significantly better than the default can be found with little to no guidance. Until better configurations are found, any evaluation terminated by the cutoff provides no guidance.
3. Use instances easier than the intended use case – in particular, instances easy enough to permit the necessary number of evaluations. There is a risk that the instances used will become so easy that they are no longer representative of the intended use case and thus misguide the configurator.

For each of these approaches, it is possible to construct examples causing failure. We found that the best configurations were often found by using easy instances in the training set [17]. However, using these same easy instances during validation resulted in good configurations being dismissed. This happened because, while easy instances provided sufficient guidance for finding good configurations on other easy instances, performance on the training set turned out to be a poor predictor for performance on the much harder testing set. Figure 1, which compares the training and testing performance for 300 configurations of LKH found by ParamILS using an easy training set, illustrates this phenomenon (see Section 4 for details on this configuration scenario).

An interesting observation from these results is that a perfect configurator – i.e., one that finds those configurations with the best training performance – would be unable to produce any of the configurations which exhibited good testing performance. In situations where training performance is either uncorrelated or, worse, negatively correlated with testing performance (as seen in Figure 1), we must rely on high variance in the underlying configurator to find configurations with good testing performance. Therefore, in the setting considered here, *a better configurator does not necessarily yield better results.*

The solution we proposed [17] was to use a training set of easy instances for the configuration process, followed by validation using a separate set of instances of intermediate difficulty to identify configurations whose performance

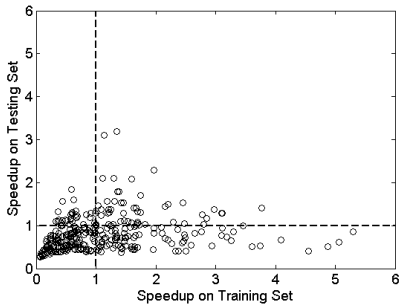

Fig. 1. A comparison between the training versus testing performance, measured as speedup over the default, for 300 configurations of LKH found by ParamILS using an easy training set. Configurations with high training speedup, near the right, are likely to be selected by the standard protocol while configurations with high testing speedups, near the top, are what we would like to find.

scales well. The fundamental idea behind our approach is that intermediate instances are more representative of the intended use case while still being feasible to use for *validation*. For the configuration scenarios we considered, intermediate difficulty was defined based on percentiles of the distribution of runtime for the default configuration of a given solver over the testing set. This configuration protocol, dubbed TE-SI, reliably outperformed the alternative approaches of performing both training and validation on easy instances (TE-SE) or on intermediate instances (TI-SI) and was able to produce configurations that were up to 180% better than the standard protocol.

This work has a similar motivation to curriculum learning [7] where training examples are presented to a learner in a schedule which progresses from generally easy examples to generally hard examples. While appearing very similar these two ideas differ significantly with respect to the definition of difficulty. For curriculum learning a difficult example is one that is hard to incorporate into a model for a given stage of the learning process; easy examples are used early in the training process because they are more informative at that stage. Where as we propose a resource-centric definition of difficulty where the difficulty of an instances is defined with respect to the resources used by some reference solver. Easy instances, despite being *less* informative, are used during the early stages of training because it is infeasible to use anything harder. The schedule of examples produced by these two ideas can be very different from each other and in the extreme the schedules produced by our methods can be the reverse of those produced by curriculum learning.

3 Validation using racing

The basic idea behind racing, as applied to algorithm configuration, is to sequentially evaluate a set of candidate configurations of a given target algorithm on a set of problem instances, one of each is presented in each stage of the race, and

to eliminate configurations from further consideration once there is sufficient evidence that they are performing significantly worse than the current leader of the race, i.e., the configuration with the best performance on the instances seen so far. The race ends when either a single configuration remains, when all problem instances have been used, or when an overall time budget has been exhausted. When racing terminates with more than one configuration remaining, the one with the best performance is selected as the final result. There are three important aspects to racing strategies: (1) how the set of candidate configurations is constructed, (2) what metric is used to determine the best configuration, and (3) what method is used to determine if a configuration can be eliminated from further consideration.

The first and most prominent racing procedure for algorithm configuration is F-Race [8], which also provides the conceptual basis for the more powerful *Sampling F-Race* and *Iterated F-Race* procedures [6]. F-Race uses the non-parametric, rank-based Friedman test to determine when to eliminate candidate configurations. A major limitation of this basic version of F-Race stems from the fact that in the initial steps, all given configurations have to be evaluated. This property of basic F-Race severely limits the size of the configuration spaces to which the procedure can be applied effectively - particularly, when dealing with initial sets of configurations obtained from so-called full factorial designs, which contain all combinations of values for a set of discrete (or discretised) parameters. This limitation is removed in Sampling F-Race, which builds a candidate set by uniformly sampling from the set of all possible configurations, and Iterative F-Race, which alternates sampling configurations from more sophisticated probabilistic models and stages of standard F-race. In contrast, we apply racing to sets of configurations determined from multiple independent runs of a powerful configuration procedure, here: ParamILS and SMAC. Basic F-Race and its variants select the instance used to evaluate configurations for each round of the race at random from the given training set. As motivated in the following, we consider the training instances in order of increasing difficulty for the default configuration of the given target algorithm.

Slow racers make for slow races. In each round of a race, every candidate configuration must be evaluated. If the majority of candidate configurations have poor performance, then much time is spent performing costly evaluations of bad configurations before anything can be eliminated. This is problematic, because good configurations are often quite rare, so that the majority of configurations in the initial candidate set are likely to exhibit poor performance. Therefore, we perform racing on a set of candidate configurations obtained from multiple runs of a powerful configurator rather than for the configuration task itself; this way, we start racing from a set of configurations that tend perform to well, assuming the configurator being used is effective, which significantly speeds up the racing process.

It doesn't take a marathon to separate the good from the bad. The first few stages of racing are the most expensive. Yet, during this initial phase, there is not yet enough information to eliminate any of the configurations, so the entire initial candidate set is being considered. We know how the default configuration of an algorithm performs on each validation instance, which gives us an idea for

the difficulty of the instance for all other configurations of the target algorithm. By using instances in ascending order of difficulty, we reserve the most difficult (i.e., costly) evaluations for later stages of the race, when there are the fewest configurations left to be evaluated.

Judge the racers by what matters in the end. The configuration scenarios examined in this work all involve minimising a given target algorithm's run time (using penalised average run time to account for timeouts). While rank-based methods, such as F-Race, may indirectly lead to a reduction in runtime (e.g., aggregate rank likely corresponds to overall runtime) they are more appropriate for scenarios where the magnitude of performance differences does not matter. We therefore propose the use of a standard permutation test instead of the rank-based Friedman test, focused on runtime, as the basis for eliminating candidate configurations.

In detail, our testing procedure works as follows. Given n configurations $c_1, \ldots c_n$, and m problem instances $i_1, \ldots, i_m$ considered at stage m of the race, we use $p_{k,j}$ to denote the performance of configuration c_k on instance i_j, and p_k to denote the aggregate performance of configuration c_k over $i_1, \ldots, i_m$. In this work, we use penalised average run time, PAR10, described in more detail in the following section, to measure aggregate performance, and our goal is to find a configuration with minimal PAR10. Let c_1 be the current leader of the race, i.e., the configuration with the best aggregate performance among $c_1, \ldots, c_n$, We now perform pairwise permutation tests between the leader, c_1, and all other configurations c_k (challengers). Each of these tests assesses whether c_1 performs significantly better than c_k; if so, c_k is eliminated from the race. To perform this one-sided pairwise permutation test between c_1 and c_k, we generate many resamples of the given performance data for these two configurations. (In our experiments, we used 100 000 resamples, since the computational cost of resampling is very low compared to that of target algorithm runs.) Each resample is generated from the original performance data by swapping the performance values $p_{1,j}$ and $p_{k,j}$ with probability 0.5 and leaving them unchanged otherwise; this is done independently for each instance $j = 1, \ldots, m$. We then consider the distribution of the aggregate performance ratios p'_1/p'_k over these resamples and determine the q-quantile of this distribution that equals the p_1/p_k ratio for the original performance data. Finally, if, and only if, $q > \alpha_2$, where α_2 is the significance of the one-sided pairwise test, we conclude that c_1 performs significantly better than c_k. Different from F-race, where the multi-way Friedman test is used to gate a series of pairwise post-tests, we only perform pairwise tests and therefore need to perform multiple testing correction. While more sophisticated corrections could in principle be applied, we decided to use the simple, but very conservative Bonferroni correction and set $\alpha_2 := \frac{\alpha}{n-1}$ for an overall significance level α.

We refer to the racing procedure that considers problem instances in order of increasing difficulty for the default configuration of the given target algorithm and in each stage eliminates configurations using the previously described series of pairwise permutation tests as *ordered permutation race (op-race)*, and the variant of basic F-race that uses the same instance ordering as *order F-race (of-race)*.

The TE-FRI and TE-PRI configuration protocols. We now return to the application of racing in the context of a configuration protocol that starts from a set of configurations obtained from multiple independent runs of a configurator, such as ParamILS. In this context, we start op-race and of-race from the easiest intermediate difficulty instance and continue racing with increasingly difficult instances until either a single configurations remains, the time budget for validation has been exhausted, or all available intermediate instances have been used.

This yields two new protocols for using algorithm configurators: (1) train-easy validate-intermediate with of-race (TE-FRI) and (2) train-easy validate-intermediate with op-race (TE-PRI). We have observed that both protocols are quite robust with respect to the significance level α (see Section 5 and generally use $\alpha = 0.01$ for TE-FRI and $\alpha = 0.1$ for TE-PRI.

4 Experimental Setup and Protocol

When assessing the configuration approaches considered in this work, we focused on two questions: (1) How well can a given approach be expected to perform on a given configuration scenario, and (2) how effective is it across different configuration scenarios. In the following, we describe the computational experiments we designed to investigate these questions. For all experiments we measure the performance of configurations on a given instance using penalised average runtime required for reaching the optimal solution and a penalty factor of 10 times the scenario-specific cutoff for every run that failed to reach the optimal solution (i.e., we measured the PAR-10 scores). The training, validation, and testing sets used for all scenarios are disjoint.

Evaluating configuration protocols. A single, randomized, configuration experiment (i.e., set of configurator runs and the corresponding global validation step) should be expected to be somewhat indicative of how the procedure used for the experiment performs in general. However, because the configurators we used here (as well as all other automated algorithm configuration procedures we are aware of) are randomised, the result of any single configuration experiment can be misleading. We therefore performed a large number of configurator runs – between 100 and 300 per unique training set – for each scenario, and fully evaluated the configuration found by each on the training, validation and testing sets. We then used these large sets of configuration runs to simulate configurator runs by randomly sampling, with replacement, a set of configurator runs of the desired cardinality; we then performed the selection procedure for the protocol under consideration by looking up the required validation results to obtain the winning configuration from each of these resampled sets. Finally, we evaluated this winning configuration on the testing set. For each scenario, protocol and target number of configurator runs , we repeated this process 100 000 times and calculated the median and [10%,90%] confidence intervals for the performance of the winning configurations from the resulting bootstrap distribution, using the standard percentile method.

TSP solving using LKH. The first scenario we considered for evaluating configuration protocols involves Keld Helsgaun's implementation of the Lin-

Kerninghan algorithm (LKH), the state-of-the art incomplete solver for the traveling salesperson problem (TSP) [12] which is able to find optimal or near optimal solutions considerably faster than the state-of-the-art complete solver Concorde [5]. In particular, we were interested in configuring LKH to solve structured instances similar to those found in the well known TSPLIB benchmark collection [16], a well studied heterogeneous set of industrial and geographic instances. From previous work [17], we know the standard protocol for using ParamILS to be very ineffective for this scenario, resulting in configurations which are significantly worse than the default. This makes this scenario of particular interest when examining new configuration procedures.

The original TSPLIB benchmark set contains only 111 instances; since we considered this too small to allow for effective automated configuration and evaluation, we generated new TSP instances based on existing TSPLIB instances by selecting 10%, 20%, or 30% of the cities uniformly at random to be removed. These 'TSPLIB-like' instances retain most of the original structure and are comparable in difficulty to the original instance, ranging from requiring a factor of 30 less time to a factor of 900 more time for the default configuration of LKH to solve. We generated 3192 instances, containing up to 6000 cities, which take a few seconds to a few hours for the default configuration of LKH to solve.

Because LKH is an incomplete solver, any particular run of LKH is not guaranteed to find an optimal solution of a given TSP instance. When evaluating and optimising the performance of LKH, we measured the runtime required by LKH to reach a target solution quality (tour length) determined by a single run of the default configuration on the given instance. For the original TSPLIB instances, we compared these target solution qualities against known optimal solution qualities; for small instances ($\leq$ 1500 cities), our target solution qualities turned out to be optimal, and for the larger instances, they were generally within 1% of the known optima.

We configured LKH for minimised PAR-10 using multiple independent ParamILS and SMAC runs of 24 CPU hours each. Overall, we performed 300 such runs on a training set of easy instances using a 120 second per-instance cutoff, where easy instances were selected from our set of TSPLIB-like instances such that each of them could be solved at least 1000 times by configurations of LKH with the same (average) performance as the default within a ParamILS run of 24 CPU hours; the rationale for this was to ensure that each ParamILS run could be expected to consider a reasonable number of LKH configurations. We also performed 100 ParamILS runs of 24 CPU hours on a training set of intermediate instances using a 780 second per-instance cutoff, where intermediate instances were defined as being in the 12.5 to 20 percentile difficulty of the testing set, which turned out to correspond to a running time between 350 and 580 CPU seconds of LKH's default configuration. A per-instance cutoff of 2 hours was used during validation and testing.

SAT solving using Lingeling. The boolean satisfiability problem (SAT), which is concerned with determining if there exists an assignment of variables which satisfies some boolean formula, is one of the most widely studied combinatorial optimization problems and has numerous industrial applications including formal verification of both software and hardware. This scenario considers con-

figuring Lingeling [2], a state-of-the-art SAT solver, for solving industrial SAT instances drawn from the 2003, 2004, 2005, 2007, 2009 and 2011 SAT competitions and the 2006, 2008 and 2010 SAT races [3]. Lingeling is a highly parameterized (i.e., 117 parameters, though most of these parameters are concerned with memory management) complete solver designed for solving industrial instances. The parallel variant of Lingeling, PLingeling [2], won a gold medal in the parallel application (i.e., industrial) track of the 2011 SAT Competition[3].

We configured Lingeling for minimised PAR10 using multiple independent ParamILS runs of 48 CPU-Hours. We performed 100 such runs for a set of easy training instances using a 180 second per-instance cutoff, where easy instances were selected to be solved by the default configuration of Lingeling within 1 to 173 seconds, which corresponds to at least 1000 successful runs within the time budget for a single configurator run. Furthermore, we performed 100 independent ParamILS runs of 48 CPU hours each on a set of training instances of intermediate difficulty using a 1200 second per-instance cutoff, where intermediate instances were defined as being in the 12.5 to 20 percentile difficulty of the testing set, which turned out to correspond to a running time between 490 and 820 CPU seconds of Lingeling's default configuration. A per-instance cutoff of 3 hours was used during validation and testing.

MIP solving using CPLEX. CPLEX is one of the best-performing and most widely used industrial solvers for mixed integer programming (MIP) problems. It is based on a highly parameterized branch-and-cut procedure that generates and solves a large number of linear programming (LP) subproblems. While most details of this procedure are proprietary, at least 76 parameters which control CPLEX's performance while solving MIP problems are exposed to end users.

We considered three scenarios for configuring CPLEX. The first two scenarios involve configuring different versions of CPLEX, 12.1 and 12.3, for a set of 2000 instances based on real data modeling wildlife corridors for grizzly bears in the Northern Rockies [11]. Hutter et al. [13] have obtained a 52× fold speedup over the default configuration on these so-called CORLAT instances when configuring CPLEX 12.1, the most recent version of CPLEX available at the time of their publication, using ParamILS with the standard configuration protocol. Our first CPLEX experiment replicates their work by using independent 20-CPU-hour runs of ParamILS. While Hutter *et al.* originally used 48-CPU-hour runs for these experiments we are using an execution environment for which CPLEX 12.1 runs nearly 2.4 times faster. We performed 100 such configurator runs on a set of easy training instances using a 300 second per-instance cutoff, where easy instances were those that took between 1 and 173 seconds for the default configuration of CPLEX 12.1 to solve, so that each instance could be solved at least 1000 times during a single run of ParamILS. We also performed 100 independent ParamILS runs of 20 CPU hours each on a set of training instances of intermediate difficulty using a 1200 second per-instance cutoff, where intermediate instances were defined as being in the 12.5 to 20 percentile difficulty of the testing set, which turned out to correspond to a running time between 660 and 1100 CPU seconds of CPLEX 12.1's default configuration. A per-instance cutoff of 2 hours was used during validation and testing.

Our second CPLEX scenario differs from the first scenario in that it uses the newer CPLEX version 12.3, whose default configuration generally tends to perform substantially better than the default configuration of CPLEX 12.1; on the CORLAT instances, the speedup of CPLEX 12.3 (default) *vs* CPLEX 12.1 (default) turned out to be nearly 20-fold. To preserve the ratio between the time required to solve training instances and the overall time budget per configuration run, we scaled the latter by a factor of 1/20 (i.e., we used ParamILS and SMAC runs of 3456 CPU seconds rather than 20 CPU hours).

Our third CPLEX scenario considered configuring CPLEX 12.3 for a set of instances based on real data modeling the spread of endangered red-cockaded woodpeckers based on decisions to protect certain parcels of land. Similar to the first scenario, we used a time budget of 48 CPU hours for each independent run of ParamILS. We performed 100 such runs for a set of easy training instances using a 180 second per-instance cutoff, where easy instances were selected to be solved by the default configuration of CPLEX 12.3 within 1 to 173 seconds, which corresponds to at least 1000 successful runs within the time budget for a single configurator run. Furthermore, we performed 100 independent ParamILS runs of 48 CPU hours each on a set of training instances of intermediate difficulty using a 1200 second per-instance cutoff, where intermediate instances were defined as being in the 12.5 to 20 percentile difficulty of the testing set, which turned out to correspond to a running time between 540 and 900 CPU seconds of CPLEX 12.3's default configuration. A per-instance cutoff of 10 hours was used during validation and testing.

Execution environment. All our computational experiments were performed on the 384 node DDR partition of the Westgrid Orcinus cluster; Orcinus runs 64-bit Red Hat Enterprise Linux Server 5.3, and each DDR node has two quad-core Intel Xeon E5450 64-bit processors running at 3.0 GHz with 16GB of RAM.

5 Results

Using the methods described in Section 4 we evaluated each of the four configuration protocols on all five configuration scenarios. The results are shown in Table 1, where we report bootstrapped median quality (in terms of speedup over the default configurations, where run time was measured using PAR10 scores) of the configurations found within various time budgets as well as bootstrap [10%,90%] percentile confidence intervals (i.e., 80% of simulated applications of the respective protocol, obtained by subsampling as explained in the previous section, fall within these ranges; not that these confidence intervals are *not* for median speedups, but for the actual speedups over simulated experiments).

As can be seen from these results, TE-PRI is the most effective configuration protocol, followed by TE-FRI and TE-SI. These three protocols tend to produce very similar [10%, 90%] confidence intervals, but the two racing approaches achieve better median speedups, especially for larger time budgets.

To further investigate the performance differences between the protocols, we compared them against a hypothetical protocol with an **oracle** selection mechanism. This mechanism uses the same configurator runs as the other protocols,

Table 1. Speedups obtained by the configuration protocols, using ParamILS, on configuration scenarios with different overall time budgets. An increase in overall configuration budget corresponds to an increase in the number of configuration runs performed rather than an increase in the time budget for individual runs of the configurator. This means larger time budgets can be achieved by increased either wall-clock time or the number of concurrent parallel configurator runs. The highest median speedups, excluding the oracle selector, for each configuration scenario and time budget are boldfaced.

	Median [10%, 90%] Speedup (PAR10)			
Time Budget (CPU Days)	TE-SI	TE-FRI	TE-PRI	Oracle Selector
Configuring LKH for TSPLIB, using ParamILS				
20	1.33 [0.96, 2.29]	**1.34** [1.00, 2.11]	**1.34** [0.95, 2.11]	1.71 [1.33, 3.11]
50	1.52 [1.06, 3.10]	1.60 [1.25, 3.10]	**1.85** [1.25, 3.10]	2.11 [1.46, 3.19]
100	2.10 [1.24, 3.19]	2.11 [1.46, 3.19]	**2.29** [1.38, 3.19]	2.29 [1.85, 3.19]
Configuring LKH for TSPLIB, using SMAC				
20	0.99 [0.71, 1.23]	1.00 [0.73, 1.23]	**1.08** [0.89, 1.23]	1.12 [0.89, 1.25]
50	**1.08** [0.89, 1.23]	**1.08** [0.92, 1.23]	**1.08** [0.89, 1.23]	1.23 [1.08, 1.25]
100	1.08 [0.89, 1.23]	**1.23** [1.00, 1.23]	**1.23** [0.89, 1.25]	1.25 [1.23, 1.25]
Configuring Lingeling for SAT-Competition and SAT Race, using ParamILS				
40	**1.00** [0.97, 1.01]	**1.00** [0.97, 1.01]	0.98 [0.93, 1.00]	1.01 [1.00, 1.08]
100	**1.00** [0.98, 1.01]	**1.00** [0.98, 1.01]	0.99 [0.94, 1.00]	1.03 [1.00, 1.08]
200	1.00 [0.98, 1.01]	1.00 [0.98, 1.01]	**1.01** [0.98, 1.08]	1.08 [1.02, 1.08]
Configuring CPLEX 12.3 for RCW, using ParamILS				
40	1.11 [0.97, 1.39]	**1.12** [0.96, 1.39]	1.08 [0.98, 1.42]	1.23 [1.08, 1.42]
100	1.12 [1.03, 1.42]	**1.16** [1.06, 1.42]	**1.16** [0.98, 1.42]	1.39 [1.16, 1.42]
200	1.13 [1.11, 1.42]	1.37 [1.06, 1.42]	**1.42** [0.98, 1.42]	1.42 [1.37, 1.42]
Configuring CPLEX 12.3 for RCW, using SMAC				
40	**0.79** [0.54, 1.01]	**0.79** [0.54, 1.24]	**0.79** [0.54, 1.01]	0.95 [0.77, 1.24]
100	0.79 [0.77, 1.24]	**0.84** [0.54, 1.24]	0.82 [0.77, 1.24]	1.01 [0.84, 1.24]
200	0.79 [0.77, 1.24]	0.84 [0.54, 1.24]	**1.24** [0.77, 1.24]	1.24 [0.98, 1.24]
Configuring CPLEX 12.1 for CORLAT, using ParamILS				
40	54.5 [42.2, 61.1]	53.8 [42.9, 61.1]	**55.8** [48.3, 61.1]	60.0 [48.8, 68.3]
100	60.1 [49.0, 68.3]	60.6 [53.4, 68.3]	**61.1** [50.3, 68.3]	61.3 [60.0, 68.3]
200	61.5 [53.8, 68.3]	**68.3** [60.1, 68.3]	**68.3** [60.6, 68.3]	68.3 [60.6, 68.3]
Configuring CPLEX 12.3 for CORLAT, using ParamILS				
1.0	2.00 [1.02, 2.64]	1.93 [1.19, 2.64]	**2.24** [1.00, 3.04]	2.36 [1.94, 3.04]
2.5	**2.36** [1.95, 3.04]	**2.36** [1.95, 3.04]	**2.36** [1.93, 3.04]	2.64 [2.24, 3.04]
5.0	2.64 [2.24, 3.04]	**3.02** [1.95, 3.04]	**3.02** [2.24, 3.04]	3.04 [2.64, 3.04]
Configuring CPLEX 12.3 for CORLAT, using SMAC				
1.0	2.41 [1.46, 3.66]	2.41 [1.39, 3.66]	**2.89** [1.54, 3.66]	2.89 [2.16, 3.84]
2.5	**3.26** [1.94, 3.84]	**3.26** [2.19, 3.84]	**3.26** [2.41, 3.66]	3.66 [2.93, 3.84]
5.0	**3.66** [2.89, 3.84]	**3.66** [3.26, 3.84]	**3.66** [2.41, 3.66]	3.84 [3.66, 3.84]

but always selects the configuration from this set that has the best *testing* performance, without incurring any additional computational burden. This provides a upper bound of the performance (speedup) that could be achieved by *any* method for selecting from a set of configurations obtained for a given training set, configurator and overall time budget. These results, shown in Table 1 and

Table 2. Impact of (a) significance level α and (b) instance ordering on the results obtained from TE-PRI and TE-FRI on LKH for TSPLIB using ParamILS. The highest median speedups for each time budget and (a) *alpha* or (b) instance ordering are boldfaced.

(a) Impact of Alpha

	Median Speedup			
Time Budget	$\alpha =$			
(CPU Days)	0.01	0.025	0.05	0.1
TE-FRI				
20	1.34	**1.38**	**1.38**	1.33
50	**1.60**	1.55	1.52	1.56
100	**2.11**	2.08	2.08	2.08
TE-PRI				
20	1.33	1.33	**1.36**	1.34
50	1.60	1.71	**2.09**	1.85
100	2.11	2.11	**2.29**	**2.29**

(b) Impact of Ordering

	Median Speedup	
Time Budget	Instance Ordering	
(CPU Days)	Ascending	Random
TE-FRI		$\alpha = 0.01$
20	**1.34**	1.29
50	**1.60**	1.57
100	**2.11**	2.10
TE-PRI		$\alpha = 0.1$
20	**1.34**	1.31
50	**1.85**	1.71
100	**2.29**	2.11

Figure 2, demonstrate that for some scenarios (e.g., CPLEX 12.1 for CORLAT) the various procedures, particularly TE-PRI, provide nearly the same performance as the oracle already while for others (e.g., CPLEX 12.3 for RCW), there is a sizable gap for a broad range of overall time budgets. Furthermore, the two racing protocol as well as TE-SI, given a sufficiently high time budget, find the same best configuration from the overall, large set as the oracle. Finally, TE-PRI tends to identify good configurations earlier than TE-FRI, particularly for small overall time budgets, where it is the only configuration protocol whose efficacy is consistently close to the bound provided by the oracle.

As can be seen in Table 1 and Figure 2(h), all protocols, including the oracle selector, failed to produce significant improvements over the default configuration of Lingeling on SAT-Competition instances. We note that the default configuration of Lingeling has been extensively hand-tuned by its developer for performance on the SAT-Competition instances used in our own experiments, and we suspect that there are few or no configurations that perform significantly better on those instances. Regardless of the overall poor performance, the protocols introduced here (i.e., TE-PRI and TE-FRI) perform at least as well as the alternatives, and in some cases (e.g., as compared to TI-SI) significantly better.

The impact of significance level α. The performance of both racing methods depends on the value of the significance level α that controls the stringency of the test used to eliminate configurations. Table 2(a) shows the effect of varying the value of α from 0.01 to 0.1 for TE-FRI and TE-PRI on LKH for TSPLIB. Overall, variation of α within this range has only a minor impact on the efficacy of the racing protocols; at the same time, it can be easily seen that at extreme values of α, performance will degrade: for $\alpha = 0$, both racing methods are equivalent to TE-SI, and at $\alpha = 1$ every configuration other than the one that appears to be best at that time is eliminated in the first round.

The importance of instance ordering. Existing algorithm configuration procedures based on racing (in particular, F-Race and its variants) randomly select instances from their training set in each round. As stated in Section 3, our intuition was that using instances in order of increasing difficulty for the default configuration would be more efficient, since this way, the hardest instances will be used only later in the process, when weaker configurations have already been eliminated and few strong configurations remain. To test this intuition, we evaluated variants of our racing protocols that used random instance ordering on LKH for TSPLIB. The results, reported in Table 2(b), demonstrate that using sorted instance ordering reliably resulted in modestly better performance; further experiments (data not shown) in which we used validation sets spanning a larger range of instance difficulty (for the default configuration of LKH) indicated that this effect increases in magnitude with the differences in difficulties between the easiest and hardest instances used for validation.

6 Conclusion and Future Work

In this work, we have addressed the problem of using automated algorithm configuration in situations where the kind of problem instances for which performance of a given target algorithm is to be optimised are too difficult to be used directly during the configuration process. Building on the idea of selecting from a set of configurations optimised on easy training instances by validating on instances of intermediate difficulty recently, we have introduced two novel protocols for using automatic configurators by leveraging racing techniques to improve the efficiency of validation. The first of these protocols, TE-FRI, uses a variant of F-Race [8] and the second, TE-PRI, uses a novel racing method based on permutation tests. Through a large empirical study we have shown that these protocols are very effective and reliably outperform the TE-SI protocol we previously introduced across every scenario we have tested. This is the case for SMAC [14] and ParamILS [15], two fundamentall different configuration procedures (SMAC is based on predictive performance models while ParamILS performs model-free stochastic local search), which suggests that our new racing protocols are effective independently of the configurator used.

We have recently begun to explore so called *interleaved protocols* in which configuration and validation are performed multiple times in an alternating fashion (i.e., the configuration and validation phases of the protocol are interleaved). This includes ideas such as racing configurators by periodically terminating individual configuator runs when their current incumbent is significantly worse than those incumbents found by other concurrent runs. Early results have shown that interleaved protocols are able to further reduce the gap between the protocols presented in this paper and the oracle selector.

References

[1] IBM ILOG CPLEX optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/. Version visited last in October 2011.

[2] Lingeling. http://fmv.jku.at/lingeling/. Version visited last in March 2012.

[3] SAT competition. http://www.satcompetition.org/. Version visited last in March 2012.
[4] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *CP-09*, pages 142–157, 2009.
[5] D. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. Concorde TSP solver. http://www.tsp.gatech.edu/concorde.html. Version visited last in October 2011.
[6] P. Balaprakash, M. Birattari, and T. Stützle. Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. In *4th International Workshop on Hybrid Metaheuristics, Proceedings, HM 2007*, volume 4771 of *Lecture Notes in Computer Science*, pages 108–122. Springer Verlag, Berlin, Germany, 2007.
[7] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *International Conference on Machine Learning, ICML*, pages 41–48, 2009.
[8] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18, 2002.
[9] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. F-Race and Iterated F-Race: An Overview. In *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336. Springer-Verlag, 2010.
[10] M. Chiarandini, C. Fawcett, and H. H. Hoos. A modular multiphase heuristic solver for post enrolment course timetabling. In *Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling*, pages 1–6, Montréal, 2008.
[11] C. P. Gomes, W. Jan van Hoeve, and A. Sabharwal. Connections in networks: A hybrid approach. In *CPAIOR*, volume 5015 of *LNCS*, pages 303–307. Springer, 2008.
[12] K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. In *European Journal of Operational Research*, volume 126, pages 106–130, 2000.
[13] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Proc. of CPAIOR-10*, volume 6140 of *LNCS*, pages 186–202. Springer, 2010.
[14] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. 5th Intl. Conference on Learning and Intelligent Optimization (LION 5)*, volume 6683 of *LNCS*, pages 507–523. Springer-Verlag, 2011.
[15] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An Automatic Algorithm Configuration Framework. In *Journal of Artificial Intelligence Research*, volume 36, pages 267–306, October 2009.
[16] G. Reinelt. TSPLIB. http://www.iwr.uni-heidelberg.de/groups/comopt/ software/TSPLIB95. Version visited last in October 2011.
[17] J. Styles, H. H. Hoos, and M. Müller. Automatically configuring algorithms for scaling performance. In *Proc. 6th Intl. Conference on Learning and Intelligent Optimization (LION 6)*, volume 7219 of *LNCS*, pages 205–219. Springer-Verlag, 2012.
[18] D. Tompkins and H. H. Hoos. Dynamic scoring functions with variable expressions: new SLS methods for solving SAT. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 278–292, 2010.
[19] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 16–30, 2011.

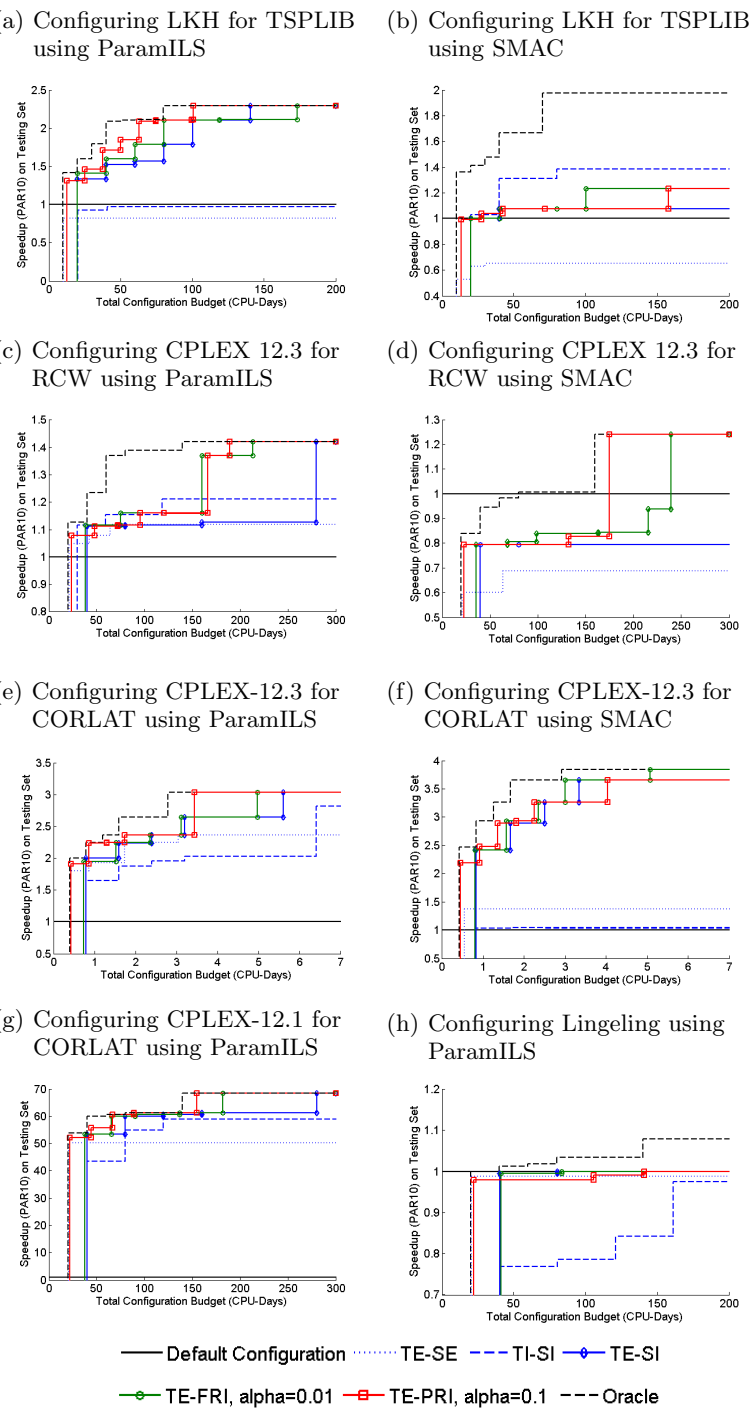

Fig. 2. Trade-off between the quality of configurations (bootstrapped median speedup in terms of PAR10 compared to default configuration) obtained by the various protocols versus the overall time budget (i.e., configuration time + validation time). The oracle indicates the best performance that can be obtained by any protocol for selecting from a set of configurations (see text for details).