Using Racing to Automatically Configure Algorithms for Scaling Performance

James Styles and Holger Hoos
University of British Columbia, Canada, e-mail: {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 are 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. An extended version of this paper can be found at www.cs.ubc.ca/labs/beta/Projects/Config4Scaling.

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, can be a challenging problem for algorithm designers. Recent work on automatically configuring algorithms has proven to be very effective. These automatic algorithm configurators rely on the use of significant computational resource to explore the design space of an algorithm.

In previous work [7] we examined a limitation of the basic protocol for using automatic algorithm configurators in scenarios where the intended use case of an algorithm is too expensive to be feasibly used during configuration. We proposed a new protocol for using algorithm configurators, referred to as train-easy select-intermediate (TE-SI), which uses so-called easy instances during the configuration step of the protocol and so-called intermediate instances during the selection step. Through a large empirical study we were able to show that TE-SI reliably out performed the basic protocol.

In this work, we show how even better configurations can be found using two novel configuration protocols that combine the idea of using intermediate instances for validation with the concept of racing. One of these protocols uses a new variant of F-Race [1] 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 protocol [7] and are able to produce configurations up to 25% better within the same time budget or 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 new protocols, we performed a empirical study across five configuration scenarios, described in Section 3. All scenarios use the freely available algorithm configurators ParamILS [5] and SMAC [4].

2 Validation using racing

Racing, as applied to algorithm configuration, evaluates a set of candidate configurations of a given target algorithm on a set of problem instances, one of which is presented in each stage of the race, and eliminates configurations from consideration once there is sufficient evidence that they are performing significantly worse than the current leader of the race. 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. There are three important aspects to racing strategies: (1) how the set of candidate configurations is constructed, (2) what metric is used to evaluate configurations, 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 [1], which 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. 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.

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 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 involve minimising a given target algorithm's runtime. While rank-based methods may indirectly lead to a reduction in runtime they are more appropriate for scenarios where the magnitude of performance differences does not matter. We therefore propose the use of a permutation test instead of the rank-based Friedman test, focused on runtime, for eliminating 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, 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. 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 100 000 resamples of the given performance data for these two configurations. 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 be applied, we decided to use the simple, but 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 ordering as *ordered 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. 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 extended version) and generally use $\alpha = 0.01$ for TE-FRI and $\alpha = 0.1$ for TE-PRI.

3 Experimental Setup and Protocol

The result of a single, randomized, configuation experiment (i.e., a set of configurator runs and the corresponding global validation step) may be misleading when trying to assess the quality of a configuation procedure. We therefore performed a large number of configurator runs, up to 300, for each scenario, and fully evaluated the configuration found by each run on the training, validation and testing sets. For a specific protocol and a target number n of configurator runs, we generated 100 000 bootstrap samples by selecting, with replacement, the configurations obtained from the n runs. For each such sample R, we chose a configuration with the selection criteria of the protocol under investigation and used the performance of that configuration on the testing set as the result of R.

For all experiments, we measured 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. For all scenarios, we configured the target algorithm for minimised PAR-10 using a set of easy training instances defined as being solvable by the default configuration within the per-instance cutoff used during training. We then defined the set of intermediate instances as being in the 12.5th to 20th percentile difficulty of the testing set. The easy, intermediate and testing instance sets are disjoint for each scenario. Each scenario can then be defined by: the target algorithm, the instance set, the configurator time budgets and the per-instance cutoffs enforced during training and testing.

TSP solving using LKH. The first scenario we considered involves configuring Keld Helsgaun's implementation of the Lin-Kerninghan algorithm (LKH), the state-of-the art incomplete solver for the traveling salesperson problem (TSP) [3], to solve structured instances similar to those found in the well known TSPLIB benchmark collection [6, 7]. Each run of ParamILS and SMAC was given a time budget of 24 hours. A 120 second per-instance cutoff was enforced during configuration and a 2 hour per-instance cutoff was enforced during testing.

MIP solving using CPLEX. The final three scenarios we considered involve configuring CPLEX, one of the best-performing and most widely used industrial solvers for mixed integer programming (MIP), for solving instances based on real data modeling either wildlife corridors for grizzly bears in the Northern Rockies [2] (CORLAT instances) or the spread of endangered red-cockaded woodpeckers based on decisions to protect certain parcels of land (RCW instances).

The first CPLEX scenario considered configuring CPLEX 12.1 for CORLAT instances. Each run of ParamILS was given a time budget of 20 hours. A 120 second per-instance cutoff was enforced during configuation and a 2 hour per-instance cutoff was enforced during testing. The second CPLEX scenario considered configuring CPLEX 12.3 for CORLAT instances. Each run of ParamILS and SMAC was given a time budget of 3456 seconds. A 15 second per-instance cutoff was enforced during configuation and a 346 second cutoff was enfored during testing. The third CPLEX scenario considered configuring CPLEX 12.3 for RCW instances. Each run of ParamILS and SMAC was given a time budget of 48 hours. A 180 second per-instance cutoff was enfored during configuration and a 10 hour cutoff was enforced during 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.

Table 1. Speedups obtained by our 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 to an increase in the time budget for individual runs of the configurator. This means larger time budgets can be achieved by increasing 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 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]

4 Results

Using the methods described in Section 3 we evaluated each of the four 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 fall within these ranges; note 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, 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 that could be achieved by *any* method for selecting from a set of configurations obtained for a given training set, configurator and time budget. These results, shown in Table 1, 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, while for others (e.g., CPLEX 12.3 for RCW), there is a sizable gap.

5 Conclusion

In this work, we have addressed the problem of using automated algorithm configuration in situations where instances in the intended use case of an algorithm 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 [1] 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 [4] and ParamILS [5], 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.

References

[1] 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.

[2] 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.

[3] K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. In *EJOR*, volume 126, pages 106–130, 2000.

[4] 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*, volume 6683 of *LNCS*, pages 507–523. Springer, 2011.

[5] 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.

[6] G. Reinelt. TSPLIB. http://www.iwr.uni-heidelberg.de/groups/comopt/ software/TSPLIB95. Version visited in October 2011.

[7] J. Styles, H. H. Hoos, and M. Müller. Automatically configuring algorithms for scaling performance. In *Proc. 6th Intl. Conference on Learning and Intelligent Optimization*, volume 7219 of *LNCS*, pages 205–219. Springer, 2012.