Configuring algorithm parameter configurators using surrogate configuration benchmarks

Nguyen Dang
KU Leuven, CODeS, imec-ITEC
Email: nguyenthithanh.dang@kuleuven.be

Thomas Stützle
IRIDIA, CoDE, Université libre de Bruxelles
Email: stuetzle@ulb.ac.be

Leslie Pérez Cáceres
IRIDIA, CoDE, Université libre de Bruxelles
Email: leslie.perez.caceres@ulb.ac.be

Patrick De Causmaecker
KU Leuven, CODeS, imec-ITEC
Email: patrick.decausmaecker@kuleuven.be

***Abstract*—Given a parameterized algorithm and a problem instance set, the task of offline automated parameter configuration consists of finding an algorithm parameter configuration that works generally well on the whole instance set in an automated manner. In recent years, many tools have been developed for solving this task. The most popular ones include ParamILS, GGA, SMAC and irace. These configurators, however, also have parameters that might have impact on the performance of the configuration process. Studying a configurator's parameters systematically on real configuration scenarios would require a huge, if feasible at all, computational cost. An alternative approach is to study the configurator on the surrogates of the real configuration scenarios. This approach has shown promising results on the configurator irace. In this work, we extend the study on irace and also configure SMAC, another configurator with a vast number of applications in practice, with surrogate configuration benchmarks.**

I. Introduction

Automated parameter configuration tools such as *ParamILS* [1], *GGA* [2], *SMAC* [3] and *irace* [4], although being designed for configuring parameterized algorithm, also have their own parameters. Currently the default values of these parameters are set manually based on the experience of the configuratiors' developers. The questions of whether or not those values are best, and how they influence the performance of the configurator on specific configuration tasks, are not trivial. If we look at a configurator itself as an optimization algorithm, we can study those questions by using any automated parameter configurator to configure its parameters, which we call here *meta-tuning*. However, it would require running many parameter configurations of the configurator on different configuration scenarios with various characteristics, which is possibly not computationally practical since each run of a parameter configuration, i.e., a run of the configurator on a configuration task, normally takes from hours to days of CPU time. This particularly holds for configurators with more than a few parameters such as GGA, SMAC and irace. One approach for solving this expensiveness is building an alternative surrogate model for each configuration task and doing the meta-tuning on the surrogates instead of the real configuration benchmarks.

The current version of irace, a popular configurator, has nine parameters. In previous work [5] we have studied them by a meta-tuning using irace itself to configure irace's parameters on seven surrogate configuration benchmarks. The meta-tuning results show statistically significant improvement over the default configuration of irace in all cases with a confidence level of 99%. The best settings of irace obtained from the meta-tuning are quite different from the default values in multiple parameters. A number of post-analyses on the performance dataset of irace given by the meta-tuning confirm the importance of those parameters and also reveal the complex interactions between them. This stresses out the necessity of such a systematic study of irace's parameters. In this work, we extend this study of irace's parameters on a number of aspects.

- In [5], the surrogate models were built from the *Empirical Performance Models* proposed in [6], which results in mixed conclusion when applying the best irace's parameter configurations obtained from the surrogate-based meta-tuning on the real configuration benchmarks. One possible reason is that the surrogates do not well represent the parts of the configuration space that the configurator is interested in. In this work, we try out the new sampling method proposed in [7] for building the surrogates which focus on those interesting regions of the configuration space.
- We apply the same technique on SMAC [3], another well-known tool for automated parameter configuration. From SMAC's manual guide [8], we can extract a list of 29 parameters which potentially have impact on the configurator's internal search procedure. Such a large number of parameters makes the choice of using surrogate benchmarks to study the configurator's parameters obvious.

II. Surrogate configuration benchmarks

A configuration benchmark consists of a target algorithm that needs to be configured, a problem instance set normally drawn from a problem instance distribution, a cost metric for measuring the quality of a run of each algorithm parameter

configuration on each problem instance, and a performance measure to aggregate the cost metric values over the whole instance set. In [5], we use seven benchmarks: four for the mixed integer programming solver CPLEX, two for the Satisfiability (SAT) solver SPEAR [9], and one for the Ant Colony Optimization algorithm for solving the Travelling Salesman Problem (ACOTSP) [10]. The cost metric for CPLEX and SPEAR benchmarks is the running time to find and prove the optimality for CPLEX, to solve the SAT problem for SPEAR; for ACOTSP, the cost metric is the solution quality obtained after a fixed amount of running time. For all benchmarks, the performance measure is the mean of the cost metric values over all problem instances. The surrogate for each benchmark is a random forest regression model proposed in [6] to predict the performance of every target algorithm parameter configuration on every problem instance. The performance dataset used to train the model is a full-matrix of 1000 random configurations $\times$ all problem instances (we limited the maximum number of instances to 1000 due to practical memory requirement). CPLEX and SPEAR's performance datasets and the random forest implementation are provided in [6] and the supplementary page http://www.cs.ubc.ca/labs/beta/Projects/EPMs/. When a parameter configuration of a configurator is run on the surrogate of a configuration benchmark, the cost of running the target algorithm is avoided since the cost metric values can be quickly obtained from the regression model, hence it significantly reduces the necessary computational cost of the configuration task and makes the meta-tuning feasible.

In [5], we have studied the consistency between the real configuration benchmarks and their surrogates by three different measures: the prediction accuracy of the regression model, the similarity on the homogeneity (*Kendall concordance coefficient (W)*) and the list of important algorithm parameters (*fANOVA* [11]) of the real and the surrogate benchmarks. These analyses show the consistency to a certain extent. However, we got mixed conclusion when testing the best irace's configurations obtained from the meta-tuning with three of the surrogates on the real benchmarks: the same improvement over irace's default configuration is observed for two benchmarks of CPLEX and SPEAR, but the configured irace is surprisingly worse than the default configuration on the benchmark of ACOTSP despite the consistency of the measures indicated by the analyses. One possible reason is that the performance dataset used to built the model is too sparely sampled on the regions of high-performance parameter configuration space, while the configurator normally exploits those regions more heavily than the rest. Another sampling method proposed in [7] for building the algorithm configuration surrogate benchmarks using trajectories of multiple configurators, which puts more focus on those good regions, is a promising alternative. The authors of [7] also generate a new set of eleven surrogate benchmarks, nine of which are for optimizing running time, and the rest are for optimizing solution quality. All of them are publicly available at http://www.ml4aad.org/algorithm-analysis/epms/.

III. SMAC AND ITS PARAMETERS

In SMAC (*Sequential Model-based Algorithm Configuration*) [3], a few number of the target algorithm configurations is firstly run on a few problem instances and a prediction model to predict the performance of every algorithm configuration on every problem instance is built based on the obtained performance dataset. This model is then used to suggest a number of algorithm configurations to be evaluated in the next step. The results obtained by actually executing these configuration are added back to the surrogate model for improving this mo del and, thus, the prediction accuracy. The update of the model and the evaluation of new algorithm configurations are alternatively called until the configuration budget is exhausted. As described in [3], the whole procedure of SMAC consists of four steps: *Initialization*, *Fit model*, *Select configurations* and *Intensify*. Here we briefly describe each step, whose relevant parameters extracted from SMAC's manual guide are listed in Table II. For a detailed description of the configurator and its parameters, the readers are referred to [3], [12] and [8].

A. Initialization

This is the starting point of the configuration procedure, in which a few algorithm runs are done in order to collect the initial performance dataset for building the regression model at the first time. In the original SMAC's paper [3], a random configuration or the default configuration of the target algorithm, if provided by the user, is run on a number of (instance, random seed) pairs. This configuration is also used as the initially incumbent, i.e, the best solution found after the initialization. In SMAC's manual guide [8], two variants for this step, namely *Iterative Capping* and *Unbiased Table* (the original initialization strategy is called *Classic*) are proposed, in which additional algorithm configurations are generated and tested against the initial incumbent. This incumbent is then updated if a better configuration is found among those configurations.

B. Fit model

In this step, a regression model based on a random forest is constructed using the performance dataset of the algorithm configuration evaluations run so far. This model predicts the performance of every algorithm configuration on every problem instance. The model is specially designed to deal with the specific characteristics of the algorithm configuration task:

- Censored data: when the cost metric is the running time, the authors of [3] have proposed a strategy called *capping* to stop a run of the target algorithm prematurely before the cut-off time is exhausted if there is evidence that the corresponding parameter configuration is not promising. This strategy can help to reduce the computational cost significantly, especially for configuration scenarios with large cut-off time. For those runs, the running time obtained is a lower bound of the real cost metric value. They are also added to the performance dataset for building the regression model but with some special imputation before being used.

- Log transformation of the performance values: when the cost metric is the running time, the random forest will predict the log-transformed value of the cost metric instead of the original one, as it was shown to give better prediction quality [13].
- Conditional parameters: the target algorithm can have conditional parameters, which are only active if their parent parameters are assigned to certain values. The random forest can take the conditionality into account when splitting at a node.

C. Select configurations

Given the prediction model built in the previous step, a number of promising parameter configurations are then generated. The quality of a configuration is evaluated using an *acquisition function*. SMAC provides different variants for this acquisition function, from the one that simply uses the performance value predicted by the model to more sophisticated ones used in the Bayesian optimization literature.

The authors use a local search to find the parameter configuration that optimizes the acquisition function value. Multiple instances of the local search, each of which returns a promising configuration, can be run. The resulting configurations are sorted according to their predicted quality. For diversification, the final list for the next step is a mix of those and random configurations.

D. Intensify

The configurations chosen in the previous step are now sequentially run by the target algorithm for a challenge against the incumbent. The comparison between two configurations is only done on (problem instance, random seed) pairs that both have been run on. The challenger is initially run on one or a few pairs that the incumbent has been run on, then its performance on those pairs is compared with the incumbent. It is eliminated if it's worse than the incumbent, otherwise, the number of pairs is doubled and the cycle of running and comparision is repeated until the challenger is either worse than the incumbent (in this case, the challenger is eliminated), or equally good or better on the whole set of (instance, seed) pairs that the incumbent has been run on (in this case, the current incumbent is replaced by the challenger).

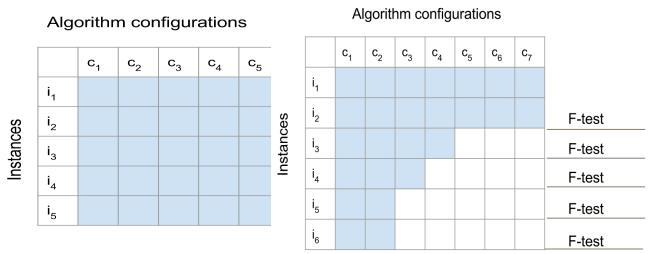
IV. IRACE AND IT'S PARAMETERS

irace exploits the idea of a machine learning technique call *racing*. Instead of running every algorithm configuration on the same set of problem instances and taking the configuration with the best performance value over the instance set as the best final configuration, in racing, the configurations are first run on a few problem instances, then a statistical test is applied to eliminate the statistically significantly worse configurations, only the remaining ones are continued to be run on another few number of instances before the statistical test is applied again to eliminate the bad configurations. As illustrated in the example in Figure 1, within the same configuration budget of 25 algorithm runs, the experiment using racing is able to see not only more algorithm configurations (hence explore better the algorithm configuration space), but also more problem instances (hence obtain better estimation of the performance of the promising configurations). irace is an iterated racing procedure. At each iteration, a race is started and run until either the configuration budget reserved for it is exhausted or only a few good configurations are remained in the race. The surviving configurations are then used to update a probabilistic model, from which new algorithm configurations for the next race (i.e., next iteration) are sampled.

Table I lists the nine parameters of irace and their types, domains and default values. Parameter μ decides how the total configuration budget is divided among different iterations. The number of instances that every configuration must be tested on at the beginning of each race before the first statistical test is applied is T^{first}. The minimum number of configurations surviving at the end of each race is set by parameter N^{min}. There are a number of possibilities for the choice of the statistical test used in the racing procedure (parameter *test_type*), including the Friedman test (F-test), the Student's t-test without the p-value correction for multiple comparisons (t-test), the Student's t-test with Bonferroni corrections (t-test-bonferroni) and the Student's t-test with Holm corrections (t-test-holm). The confidence level of the statistical test is decided by parameter *confidence_level*. In order to avoid premature convergence in the probabilistic model for generating new configurations at the beginning of each race, irace has a restart scheme for this model, which is activated when parameter *soft_restart* is enabled. At a few beginning steps of each race, it might happen that the elite configurations obtained from the previous race are eliminated too early due to their bad luck on a few tested problem instances while it does not mean that those configurations are not good. To avoid that situation, a strategy called *elitist* (parameter *enable_elitist*), which only allow eliminating elite configurations from previous race as long as the new configurations have also been run on all instances that the elites were run on so far. T^{new} is an accompanied parameter with this strategy, which decides the number of unseen problem instances all configurations need to be evaluated on (including the elite configurations from the previous race) before the new configurations are run on the instances the elites have been run on so far. The last parameter, b^{mult} is for a new version of irace, called *capping irace*. This version is specially designed for the algorithm configuration scenarios with the cost metric of running time. This new version incorporates the *adaptive capping* mechanism proposed in the two algorithm configurators ParamILS [1] and SMAC [3]. For a detailed description of irace and its parameters, the readers are referred to [14], [15] and [16].

ACKNOWLEDGMENT

This work is funded by COMEX (Project P7/36), a BELSPO/IAP Programme. A part of the computational resources and services used in this work was provided by the VSC (Flemish Supercomputer Center), funded by the Hercules Foundation and the Flemish Government - department EWI.



(a) Without racing

(b) With racing

Fig. 1: An illustration of racing technique for the task of automated parameter configuration.

Parameter	Type	Domain	Default value
μ	integer	[1, 20]	5
T^{first}	integer	[4, 20]	5
N^{min}	integer	[1, 20]	$2 + log_2 n^{params}$
test_type	categorical	F-test, t-test, t-test-bonferroni, t-test-holm	F-test
confidence_level	real	[0.5, 0.99]	0.95
enable_soft_restart	categorical	true, false	true
enable_elitist	categorical	true, false	true
T^{new}	integer	[1, 10]	1
b^{mult}	real	[1.0, 10.0]	2.0

TABLE I: Parameters of irace

REFERENCES

[1] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: an automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, pp. 267–306, Oct. 2009.

[2] C. Ansótegui, M. Sellmann, and K. Tierney, "A gender-based genetic algorithm for the automatic configuration of algorithms," in *Principles and Practice of Constraint Programming, CP 2009*, ser. Lecture Notes in Computer Science, I. P. Gent, Ed. Springer, Heidelberg, Germany, 2009, vol. 5732, pp. 142–157.

[3] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization, 5th International Conference, LION 5*, ser. Lecture Notes in Computer Science, C. A. Coello Coello, Ed. Springer, Heidelberg, Germany, 2011, vol. 6683, pp. 507–523.

[4] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari, "The irace package: Iterated racing for automatic algorithm configuration," http://iridia.ulb.ac.be/supp/IridiaSupp2016-003/, 2016.

[5] N. Dang, L. Pérez Cáceres, T. Stützle, and P. De Causmaecker, "Configuring irace using surrogate configuration benchmarks," IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, Tech. Rep. TR/IRIDIA/2017-004, February 2017.

[6] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artificial Intelligence*, vol. 206, pp. 79–111, 2014.

[7] K. Eggensperger, M. Lindauer, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Efficient benchmarking of algorithm configuration procedures via model-based surrogates," 2017.

[8] F. Hutter and S. Ramage, "Manual for smac version v2.10.03-master," http://www.cs.ubc.ca/labs/beta/Projects/SMAC/v2.10.03/manual.pdf, 2015.

[9] D. Babić and F. Hutter, "Spear theorem prover," in *SAT'08: Proceedings of the SAT 2008 Race*, 2008.

[10] T. Stützle, "ACOTSP: A software package of various ant colony optimization algorithms applied to the symmetric traveling salesman problem," 2002. [Online]. Available: http://www.aco-metaheuristic.org/aco-code/

[11] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "An efficient approach for assessing hyperparameter importance," in *Proceedings of the 31th International Conference on Machine Learning*, vol. 32, 2014, pp. 754–762. [Online]. Available: http://jmlr.org/proceedings/papers/v32/hutter14.html

[12] ——, "Bayesian optimization with censored response data," in *NIPS workshop on Bayesian Optimization, Sequential Experimental Design, and Bandits*, 2011, published online.

[13] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: portfolio-based algorithm selection for SAT," *Journal of Artificial Intelligence Research*, vol. 32, pp. 565–606, Jun. 2008.

[14] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari, "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, pp. 43–58, 2016.

[15] M. López-Ibáñez, L. Pérez Cáceres, J. Dubois-Lacoste, T. Stützle, and M. Birattari, "The irace package: User guide," IRIDIA, Université Libre de Bruxelles, Belgium, Tech. Rep. TR/IRIDIA/2016-004, 2016. [Online]. Available: http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2016-004.pdf

[16] L. Pérez Cáceres, M. López-Ibá nez, H. Hoos, and T. Stützle, "An experimental study of adaptive capping in irace," IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, Tech. Rep. TR/IRIDIA/2017-001, January 2017.

Step 1: Initialization

1) *--init-mode*: Initialization Mode
   Domain: {CLASSIC, ITERATIVE_CAPPING, UNBIASED_TABLE}. Default: CLASSIC
2) *--initial-incumbent-runs*: initial amount of runs to schedule against for the default configuration
   Domain: [1, 2147483647]. Default: 1.
   Condition: *--init-mode* = CLASSIC
3) *--doubling-capping-challengers*: Number of challengers to use with the doubling capping mechanism
   Domain: [1, 2147483647]. Default: 2
   Condition: –init-mode = ITERATIVE_CAPPING
4) *--doubling-capping-runs-per-challenger*: Number of runs each challenger will get with the doubling capping initialization strategy
   Domain: [1, 2147483647]. Default: 2
   Condition: –init-mode = ITERATIVE_CAPPING
5) *--unbiased-capping-challengers*: Number of challengers we will consider during initialization
   Domain: [1, 2147483647]. Default: 2
   Condition: --init-mode = UNBIASED_TABLE
6) *--unbiased-capping-runs-per-challenger*: Number of runs we will consider during initialization per challenger
   Domain: [1, 2147483647]. Default: 2
   Condition: --init-mode = UNBIASED_TABLE

Step 2: Fit model

7) *--exec-mode*: execution mode of the automatic configurator
   Domain: SMAC, ROAR. Default: SMAC
   --exec-mode = ROAR means that there is no prediction model used.
8) *--mask-censored-data-as-kappa-max*: Mask censored data as kappa Max (i.e., cut-off time, or cut-off time $\times$ a penalty factor if the algorithm cannot solve the problem instance within the cutoff time. The penalty factor is defined by the intra-objective function, e.g., 10 for MEAN10)
   Domain: {true, false}. Default: false. Used only if minimizing running time.
9) *--imputation-iterations*: amount of times to impute censored data when building model, i.e., the number of times censored data points are imputed and the model is re-fitted using the new imputed data.
   Domain: [1, 2147483647]. Default: 2. Used only if minimizing running time.
10) *--mask-inactive-conditional-parameters-as-default-value*: build the model treating inactive conditional values as the default value.
    Domain: {true, false}. Default: true
11) *--rf-full-tree-bootstrap*: bootstrap all data points into trees.
    Domain: {true, false}. Default: false
12) *--rf-ignore-conditionality*: ignore conditionality for building the model (while splitting at a node)
    Domain: {true, false}. Default: false
13) *--rf-impute-mean*: impute the mean value for all censored data points
    Domain: {true, false}. Default: false
14) *--rf-log-model*: store response values in log-normal form
    Domain: {true, false}. Default: true if optimizing running time, false if optimizing quality
15) *--rf-num-trees*: number of trees to create in random forest
    Domain: [0, 2147483647]. Default: 10
16) *--rf-penalize-imputed-values*: treat imputed values that fall above the cutoff time, and below the penalized max time, as the penalized max time
    Domain: {true, false}. Default: false
17) *--rf-ratio-features*: ratio of the number of features to consider when splitting a node
    Domain: (0,1]. Default: 5/6
18) *--rf-shuffle-imputed-values*: shuffle imputed value predictions between trees
    Domain: {true, false}. Default: false
19) *--rf-split-min*: minimum number of elements needed to split a node
    Domain: [0, 2147483647]. Default: 10
20) *--rf-preprocess-marginal*: build random forest with preprocessed marginal
    Domain: {true, false}. Default: true
21) *--rf-store-data*: store full data in leaves of trees
    Domain: {true, false}. Default: false

Step 3: Select configurations

22) *--acq-func*: acquisition function to use during local search
    Domain: EXPONENTIAL, SIMPLE, LCB, EI. Default: EXPONENTIAL if minimizing runtime, EI otherwise.
    Not recommended combinations: *--acq-func* == EXPONENTIAL && *--rf-log-model* == false, *--acq-func* == EI && *--rf-log-model* == true
23) *--num-challengers*: number of challengers needed for local search, i.e., the number of local searches runs started with the chosen acquisition function, each of which returns a configuration (challenger) to test against the current incumbent in the next step (Intensify)
    Domain: [0, 2147483647]. Default: 10
24) *--num-ei-random*: number of random configurations to evaluate during EI search, i.e., after all the local searches have finished, a number of random configurations is also generated and their acquisition values are evaluated, then the configurations with the best acquisition values chosen from the set of the configurations returned by the local searches and those random configurations are returned.
    Domain: [1, 2147483647]. Default: 10000
25) *--num-ls-random*: Number of random configurations that will be used as potential starting points for local search
    Domain: [0, 2147483647]. Default: 0. Condition: *--num-ls-random* $\leq$ *--num-challengers*

Step 4: Intensify

26) *--initial-challenger-runs*: initial amount of runs to request when intensifying on a challenger
    Domain: [1, 2147483647]. Default: 1
27) *--ac-add-slack*: amount to increase computed adaptive capping value of challengers by (post scaling)
    Domain: (0, ∞). Default: 1.0. Used only if minimizing running time.
28) *--ac-mult-slack*: amount to scale computed adaptive capping value of challengers by
    Domain: (0, ∞). Default: 1.3. Used only if minimizing running time.
29) *--intensification-percentage*: percent of time to spend intensifying versus model learning
    Domain: (0,1). Default: 0.5

TABLE II: SMAC's parameters extracted from SMAC's manual guide for the meta-tuning experiments.