Model-Based Algorithm Configuration with Adaptive Capping and Prior Distributions

Ignace Bleukx*, Senne Berden*, Lize Coenen, Nicholas Decleyre, and Tias Guns

KU Leuven, Belgium *firstname.lastname*@kuleuven.be

Abstract. Many advanced solving algorithms for constraint programming problems are highly configurable. The research area of algorithm configuration investigates ways of *automatically* configuring these solvers in the best manner possible. In this paper, we specifically focus on algorithm configuration in which the objective is to decrease the time it takes the solver to find an optimal solution. In this setting, *adaptive capping* is a popular technique which reduces the overall runtime of the search for good configurations by adaptively setting the solver's timeout to the best runtime found so far. Additionally, sequential model-based optimization (SMBO)—in which one iteratively learns a surrogate model that can predict the runtime of unseen configurations—has proven to be a successful paradigm. Unfortunately, adaptive capping and SMBO have thus far remained incompatible, as in adaptive capping, one cannot observe the true runtime of runs that time out, precluding the typical use of SMBO. To marry adaptive capping and SMBO, we instead use SMBO to model the probability that a configuration will *improve* on the best runtime achieved so far, for which we propose several decomposed models. These models also allow defining prior probabilities for each hyperparameter. The experimental results show that our DeCaprio method speeds up hyperparameter search compared to random search and the seminal adaptive capping approach of ParamILS.

Keywords: algorithm configuration, adaptive capping, sequential model-based optimization, prior distributions

1 Introduction

Constraint solvers are used on a daily basis for solving combinatorial optimization problems such as scheduling, packing and routing. Because of the advanced nature of constraint solvers today, the runtime of a solver on even a single problem instance can vary tremendously depending on its hyperparameter configuration.

In *algorithm configuration*, the goal is to find a well-performing hyperparameter configuration automatically. In this field, sequential model-based optimization (SMBO) is a highly successful paradigm [2, 9, 11]. Applications of SMBO

* These authors contributed equally

approaches aim to optimize a black-box function that evaluates a configuration by running a solver with that configuration and returning the runtime. These techniques sequentially approximate this function by observing runtimes during the search and updating a probabilistic estimate [2]. This model can then be exploited to direct the search towards good configurations.

However, to obtain these runtimes during the search, one is required to allow the solver to execute until completion, regardless of the quality of the configuration considered. This stands in stark contrast with *adaptive capping*, a technique for optimising runtime in which solver runs using unfruitful configurations are stopped from the moment the runtime surpasses that of the best configuration found so far [3, 4, 7].

This work is motivated by the observation that with adaptive capping in place, one is especially keen on finding good configurations sooner rather than later, as this allows decreasing the runtime timeout of all subsequent runs; thereby reducing the runtime of the search as a whole.

To marry adaptive capping with SMBO, we step away from the idea of a model that can estimate a configuration's runtime. Instead, our models are related to the probability that a configuration will be better than any of the already considered ones. More specifically, we consider probabilistic models that can be decomposed for each individual hyperparameter independently. They are easy to compute and update, thereby keeping the overhead in between solve calls small, and allowing to scale to an arbitrarily large number of hyperparameters.

One further benefit of this decomposition is that we can now define *prior probabilities* for each hyperparameter. These can overcome the cold start problem by guiding the candidate selection from the very first iteration. To the best of our knowledge, prior information is only used in SMBO within advanced machine learning models, often in the form of transfer learning techniques [14].

As a first step, we focus in this paper on hyperparameter configuration search over a finite grid of discrete hyperparameters, and on finding the best runtime for a single problem instance. Extensions to multiple instances and more are discussed in the conclusion.

Our contributions are summarized as follows:

- We propose a sequential model-based optimization (SMBO) approach with adaptive capping: DeCaprio[1];
- We propose several decomposable probabilistic models that assign a high probability to configurations that are likely to improve on the current best runtime, and that can be updated cheaply using counting-based mechanisms;
- We propose a methodology for computing independent prior distributions for each hyperparameter, based on a heterogeneous dataset of instances; and we discuss how to use these prior distributions in SMBO;
- We show that using these decomposed distributions, especially with informed prior probabilities, leads to faster hyperparameter configuration grid search with adaptive capping than both a random grid search baseline and the seminal ParamILS approach.

[1] DEcomposable adaptive CApping with PRIOrs

Algorithm 1 generic Sequential Model-Based Optimization, from [2]

function SMBO(f, M^0, T, S)
  $\mathcal{H} \leftarrow \emptyset$
  for $t \leftarrow 1$ to T **do**
    $\hat{x} \leftarrow \text{argmax}_x\ S(x, M^{t-1})$
    Evaluate $f(\hat{x})$ ▷ Expensive step
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{(\hat{x}, f(\hat{x}))\}$
    Fit a new model M^t to $\mathcal{H}$
  return $\mathcal{H}$

2 SMBO for hyperparameter configuration

Algorithm 1 shows a generic Sequential Model-Based Optimization [2]. It takes as input an unknown (black-box) function f, a model M also called the *surrogate model* that aims to estimate f based on the past evaluations $\mathcal{H}$, a maximum number of iterations T and a scoring function S, also called acquisition function, that uses M^{t-1} to evaluate how promising it is to choose a configuration $\hat{x}$ as the next configuration to run.

SMBO techniques differ in the way they instantiate the four inputs. For algorithm configuration of constraint solvers, the typical choice is to have f be the runtime, objective value or optimality gap [8]; and M a machine learning model that estimates f, such as Gaussian processes [2], tree parzen estimators [2] or random forests [9].

An effective technique from non-SMBO-based algorithm configuration is to do adaptive capping [3, 7] of the runtime, with the best runtime known so far. However, in standard SMBO, this would complicate the use of machine learning to build a model M^t of f, as one would no longer observe the real f, but a potentially truncated version of it. So instead, to marry SMBO and adaptive capping, we build a surrogate model M^t that can be used to obtain a configuration $\hat{x}$ that is likely to *improve* on the current best runtime (i.e., where $f(\hat{x}) < min_{(x,f(x))\in\mathcal{H}} f(x)$), without having to estimate $\hat{x}$'s runtime itself.

For simplicity and efficiency, we keep away from feature-based machine learning methods that can be costly to (re)train and that have many hyperparameters of their own. Instead, we further decompose the surrogate over the individual hyperparameters and use counting-based models that can be updated incrementally as explained in the next section.

Another important difference with generic SMBO is that we assume a finite grid of possible configurations $\mathcal{X}$ that can be exhaustively searched over. We mimic the property of grid search that a configuration is never visited twice, by computing $\text{argmax}_x\ S(x, M^{t-1})$ over *unseen* configurations only (i.e. configurations not in $\mathcal{H}$).

3 Surrogate models M and scoring functions S

We consider models for which we can compute M^t based on M^{t-1} and whether the just run configuration $\hat{x}$ improved the runtime or not, i.e. whether $f(\hat{x}) < min_{(x,f(x))\in\mathcal{H}} f(x)$ is true.

3.1 Hamming similarity: searching near the current best

A first scoring function S is inspired by local search algorithms like ParamILS [7]. In local search, we first explore configurations which are in the direct neighborhood of x^*, the best configuration seen so far. This allows us to define M^t as follows

$$M^t \leftarrow \hat{x} \text{ if } f(\hat{x}) \text{ improved the runtime, else } M^{t-1} \tag{1}$$

The scoring function is then the Hamming similarity between the current best configuration M^t and each of the unseen configurations x. Let H be the set of all possible hyperparameters, and let $value(h, x)$ be the value of hyperparameter $h \in H$ in configuration x. We can then write the Hamming similarity score as

$$S(x, M^t) = \sum_{h\in H} [\![value(h, x) = value(h, M^t)]\!] \tag{2}$$

where $[\![\cdot]\!]$ is the Iverson bracket which converts True/False into 1/0.

3.2 Beta distribution

In this first probabilistic scheme, we associate a probability $P^t(X = x)$ with every configuration x. Model M^t decomposes $P^t(X)$ into a separate independent Bernoulli distribution $P^t(X_{h,v})$ for every value v of every hyperparameter h, where $X_{h,v}$ is a binary random variable whose value denotes whether hyperparameter h takes value v. The score function then becomes

$$S(x, M^t) = P^t(X = x) = \prod_{h\in H} P^t(X_{h,value(h,x)} = 1) \tag{3}$$

If $X_{h,v} \sim B(p_{h,v})$, we can consider $p_{h,v}$ itself as uncertain—as is common in Bayesian statistics—and use an appropriate prior distribution for $p_{h,v}$, rather than a point probability. For this purpose, we take a beta distribution, as it is a *conjugate* prior for a Bernoulli distribution. This means that when the beta prior is updated with a Bernoulli likelihood, the resulting posterior is again a beta distribution.

A beta distribution is characterized by parameters α and β, which can be thought of as respectively representing a number of successes (i.e. improvements to the current best time) and failures (i.e. timeouts due to adaptive capping).

$$P^t(X_{h,v} = 1) = \mathrm{E}(p_{h,v}) = \frac{\alpha^t_{h,v}}{\alpha^t_{h,v} + \beta^t_{h,v}} \tag{4}$$

So, for every hyperparameter-value combination (h, v), we count the number of successes $\alpha_{h,v}$ and failures $\beta_{h,v}$, which are both initialised to 1. Updating the model M^t with respect to a chosen $\hat{x}$ then amounts to updating these counts based on whether $\hat{x}$ improved the current timeout r^{t-1} or not:

$$\alpha^t_{h,value(h,\hat{x})} \leftarrow \alpha^{t-1}_{h,value(h,\hat{x})} + [\![f(\hat{x}) < r^{t-1}]\!] \tag{5}$$

$$\beta^t_{h,value(h,\hat{x})} \leftarrow \beta^{t-1}_{h,value(h,\hat{x})} + [\![f(\hat{x}) \geq r^{t-1}]\!] \tag{6}$$

3.3 Dirichlet distribution

A disadvantage of the above approach is that by maintaining a separate beta distribution for every value, we do not utilize the knowledge that in any configuration, every hyperparameter takes only a single value.

In what follows, model M^t instead decomposes $P^t(X)$ into a separate independent categorical distribution $P^t(X_h)$ for every hyperparameter h, where X_h is a random variable that denotes the value of h. The score function then becomes

$$S(x, M^t) = \prod_{h \in H} P^t_h(X_h = value(h, x)) \tag{7}$$

For any hyperparameter h with k distinct values $v_1 \dots v_k$, the associated categorical distribution is characterized by parameters $p_{h,v_1} \dots p_{h,v_k}$. We can again consider these uncertain and use a prior distribution. For this purpose, we use a Dirichlet distribution, as it is a conjugate prior for the categorical distribution. A Dirichlet distribution is characterized by parameters $\alpha_{h,v_1} \dots \alpha_{h,v_k}$, which can again be thought of as success counts.

$$P^t_h(v) = \mathrm{E}(p_{h,v}) = \frac{\alpha^t_{h,v}}{\sum_{j=1}^{k} \alpha^t_{h,v_j}} \tag{8}$$

For every hyperparameter-value combination (h, v), we count the number of successes $\alpha_{h,v}$, which are all initialised to 1. When a considered configuration $\hat{x}$ improves the current timeout r^{t-1}, the model M^t is updated by incrementing the appropriate counts:

$$\alpha^t_{h,value(h,\hat{x})} = \begin{cases} \alpha^{t-1}_{h,value(h,\hat{x})} + 1 & \text{if } f(\hat{x}) \text{ improved the runtime} \\ \alpha^{t-1}_{h,value(h,\hat{x})} & \text{otherwise} \end{cases} \tag{9}$$

When $\hat{x}$ does not improve the current timeout, we add a success count for every *other* hyperparameter value, thereby discounting the values of $\hat{x}$.

While this score function can be used on its own, it can also be used as a tie breaker when multiple configurations have the same Hamming score. This was also considered for the beta-distribution-based model, but empirically proved to be less effective.

4 Learning priors

The previously described approaches all start with a uniform initial distribution and learn which hyperparameters are better during search. However, this leaves them with a *cold start* problem, meaning that the first choices are uninformed. However, in the adaptive capping setting, the first configurations are the most crucial, as any improvement in runtime saves time in all subsequent runs. Hence, we determine the first runtime cap using the default configuration.

We now aim to take this one step further. Our goal is to learn a *prior probability distribution* that plays well with the above models, that is, a prior distribution that is decomposed for each of the hyperparameters independently. We model each hyperparameter h's prior as a Dirichlet distribution, which is characterized by a pseudocount value $\alpha_{h,v}$ for each value v.

To determine an informed prior, we propose to make use of a heterogeneous set of problems K on which the solver can be run. We refer to this set as the *prior set*. To cope with the sensitivity of modern solvers' runtimes [5], and to account for the existence of equivalent optimal configurations, we propose not to base the priors solely on the best configuration for a model (i.e. $x^k = argmin_x f^k(x)$). Instead, to make the priors more robust, we propose to collect all configurations whose runtime is at most 5% worse than the best one. Let us denote this set as O_k for any individual problem $k \in K$:

$$O_k = \{x \in \mathcal{X} \mid f^k(x) \leq 1.05 \cdot \min_{x' \in \mathcal{X}}(f^k(x'))\} \tag{10}$$

This gives us a *set* of best configurations, on which we compute Dirichlet pseudocounts $\hat{\alpha}_{h,v}$ using the relative number of times hyperparameter h's value v was used in O_k, across all k:

$$\hat{\alpha}_{h,v} = \sum_{k \in K} \frac{|\{x \in O_k \mid value(h,x) = v\}|}{|O_k|} \tag{11}$$

The resulting pseudocounts will have a magnitude that depends on the size of K, which could make individual updates during SMBO insignificant if K is large. So, to use these pseudocounts as a Dirichlet prior, we first normalize and scale them as $\alpha_{h,v} = |Values(h)| \cdot \hat{\alpha}_{h,v} / \sum_{v' \in Values(h)} \hat{\alpha}_{h,v'}$, where $Values(h)$ denotes the set of all possible values of hyperparameter h.

5 Experiments

In this section, we empirically answer the following research questions:

Q1 How do the different scoring functions compare in their ability to find good configurations quickly?
Q2 Does the use of an informed prior improve the ability to find good configurations quickly?
Q3 How does DeCaprio perform against ParamILS, a seminal hyperparameter configurator?

5.1 Experimental setup

All algorithms were implemented in Python3 and run on a Intel Xeon 4214 CPU. We used a collection of 190 heterogeneous CP models[2] modelled in the CPMpy library [6]. They have a runtime[3] ranging from 1ms to 2s when solved using the default parameters of OR-Tools' CP-SAT Solver v9.1 [13]. On each of the instances, we configured 9 CP-SAT hyperparameters, totalling to a grid of 13608 configurations. All code, models and data are available in the repository accompanying this paper.[4]

For the surrogates that use an informed prior, we use leave-one-out cross validation to split the models into prior and test set. To compare results across different models, we introduce a metric that denotes the relative improvement of the timeout. Here, the default's runtime has value 0, while value 1 denotes the runtime of the optimal configuration. All results are averages over all 190 instances and 10 random seeds.

5.2 Comparison of models and surrogates

We want to find good configurations as soon as possible. In Figure 1, this corresponds to an early rising curve. As baseline we use a uniform random search. From Figure 1 (left), one can conclude that most surrogates perform more or less equally at the start of the search. However, further into the search, both Hamming and Hamming + Dirichlet tiebreaker surrogates clearly outperform the alternatives. This can also be seen in Table 1. This answers **Q1**.

Table 1: Rel. improvement on full grid

Iteration	10	10^2	10^3	10^4
Random uniform	0.67	0.87	0.95	**1.00**
Hamming	0.56	0.92	**0.99**	**1.00**
Beta	0.63	0.87	0.97	**1.00**
Dirichlet	0.64	0.88	0.95	**1.00**
Hamm + D	0.58	0.93	**0.99**	**1.00**
Sample prior	0.79	0.92	0.98	**1.00**
Sorted prior	0.74	0.87	0.98	**1.00**
Dirichlet prior	0.81	0.87	0.94	**1.00**
Hamm + D prior	**0.82**	**0.94**	**0.99**	**1.00**

Using the same metric, we analyze the performance of the surrogates with an informed prior. We introduce two additional baseline algorithms: *Sample prior* and *Sorted prior*. These respectively sample and take the argmax of the prior probabilities. Neither algorithm updates its model during the search. The results of these baseline algorithms clearly show the importance of a good update rule. Figure 1 (right) and Table 1 show that using an informed prior clearly yields better performance early in the search. This answers **Q2**.

5.3 Comparison with ParamILS

We cannot compare to SMBO techniques like SMAC [10] as they do not support adaptive capping and hence would have unwieldy large total runtimes. ParamILS

[2] From Håkan Kjellerstrand's collection: http://www.hakank.org/cpmpy/
[3] The number of threads was limited to 1 for every solver call
[4] https://github.com/ML-KULeuven/DeCaprio

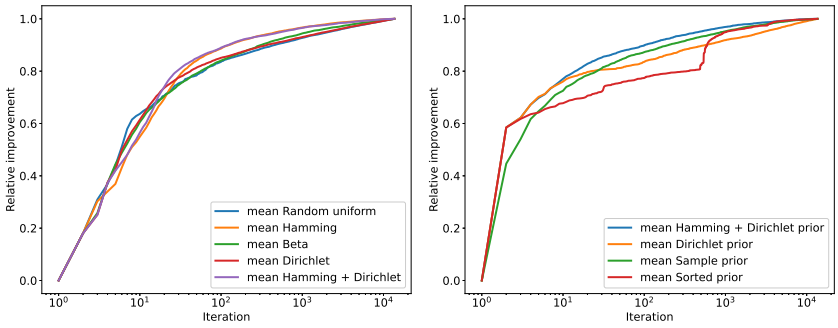

Fig. 1: Relative improvement per iteration. Left: no prior, Right: informed prior

is a seminal algorithm configuration method supporting adaptive capping [7]. Still, its computational overhead is significant on small instances so we have to compare on a smaller grid of 200 configurations. We use the same settings used by the original authors and calculate the relative improvement based on the global best runtime found.

The results of this experiment are summarized in Table 2. Here, we notice that ParamILS performs about equally well as the uninformed Hamming + Dirichlet. When comparing with an *informed* prior, however, our method obtains notably better performance in the first iterations (and hence in the total solve time, not shown).

Table 2: Small grid

Iteration	10	50	100	200
Random uniform	0.71	0.94	0.97	**1.00**
Hamm + D	0.77	**0.98**	**0.99**	**1.00**
Hamm + D prior	**0.89**	**0.98**	**0.99**	**1.00**
ParamILS	0.76	**0.98**	**0.99**	**1.00**

6 Conclusion and future work

In this paper we proposed a scheme for marrying SMBO-based approaches with adaptive capping in the context of algorithm configuration, by using decomposable models and surrogates related to the probability of a runtime improvement.

To make more informed decisions in early iterations we show how to obtain and use Dirichlet *priors* per hyperparameter. Using such prior knowledge showed big improvements over all other methods. Their decomposed nature also means they are easy to extend and combine with new hyperparameters and values, and we envision solver developers to precompute these on their own benchmarks.

In general, algorithm configuration techniques, even SMBO ones, make extensive use of sampling and our light-weight prior-based approach can replace or be combined with current (often random) sampling schemes used, potentially guiding the methods better in their early iterations.

Many directions for future work remain, including evaluating our method on larger datasets with more solvers and models and extending it to configuration over multiple instances, where racing and successive halving [1, 12] are orthogonal techniques that can be combined with the ideas presented in this paper.

Acknowledgments This research was partly funded by the Flemish Government (AI Research Program), the Research Foundation - Flanders (FWO) projects G0G3220N and S007318N and the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (Grant No 101002802, CHAT-Opt).

References

1. Anastacio, M., Hoos, H.: Model-based algorithm configuration with default-guided probabilistic sampling. In: International Conference on Parallel Problem Solving from Nature. pp. 95–110. Springer (2020)
2. Bergstra, J., Bengio, Y.: Algorithms for hyper-parameter optimization. In: In NIPS. pp. 2546–2554 (2011)
3. Cáceres, L.P., López-Ibáñez, M., Hoos, H., Stützle, T.: An experimental study of adaptive capping in irace. In: International Conference on Learning and Intelligent Optimization. pp. 235–250. Springer (2017)
4. De Souza, M., Ritt, M., López-Ibáñez, M.: Capping methods for the automatic configuration of optimization algorithms. Computers & Operations Research p. 105615 (2021)
5. Fichte, J.K., Hecher, M., McCreesh, C., Shahab, A.: Complications for computational experiments from modern processors. In: 27th International Conference on Principles and Practice of Constraint Programming (CP 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2021)
6. Guns, T.: Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In: Proceedings of the 18th workshop on Constraint Modelling and Reformulation, held with CP. vol. 19 (2019)
7. Hutter, F., Hoos, H., Leyton-Brown, K., Stützle, T.: Paramils: An automatic algorithm configuration framework. J. Artif. Intell. Res. (JAIR) **36**, 267–306 (10 2009)
8. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Automated configuration of mixed integer programming solvers. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming. pp. 186–202. Springer (2010)
9. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Coello, C.A.C. (ed.) LION. Lecture Notes in Computer Science, vol. 6683, pp. 507–523. Springer (2011)
10. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Coello, C.A.C. (ed.) Learning and Intelligent Optimization. pp. 507–523. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
11. Kerschke, P., Hoos, H.H., Neumann, F., Trautmann, H.: Automated algorithm selection: Survey and perspectives. Evolutionary Computation **27**(1), 3–45 (03 2019). https://doi.org/10.1162/evco_a_00242
12. López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L.P., Birattari, M., Stützle, T.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives **3**, 43–58 (2016)
13. Perron, L., Furnon, V.: Or-tools, https://developers.google.com/optimization/
14. Yogatama, D., Mann, G.: Efficient transfer learning method for automatic hyperparameter tuning. In: Artificial intelligence and statistics. pp. 1077–1085. PMLR (2014)

A Adapted SMBO

Algorithm 2 our Sequential Model-Based Optimization

function SMBO($f, M^0, \mathcal{X}, S, x^{\text{default}}, \hat{T}$)
  $\hat{x} \leftarrow x^{\text{default}}$
  $r \leftarrow f(x^{\text{default}})$
  $\mathcal{H} \leftarrow \emptyset$
  $T \leftarrow min(|\mathcal{X}|, \hat{T})$
  for $t \leftarrow 1$ to T **do**
    $\hat{x} \leftarrow \text{argmax}_{x \in X \setminus \mathcal{H}}\ S(x, M^{t-1})$
    Evaluate $f(\hat{x})$, cap runtime at r ▷ Expensive step
    Fit a new model M^t by updating M^{t-1} with $(\hat{x}, [f(\hat{x}) < r])$
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{\hat{x}\}$
    $r \leftarrow min(r, f(\hat{x}))$
  return $\mathcal{H}$