# MATE: A Model-based Algorithm Tuning Engine - A proof of concept towards transparent feature-dependent parameter tuning using symbolic regression

Mohamed El Yafrani, Marcella Scoczynski, Inkyung Sung, Markus Wagner, Carola Doerr, Peter Nielsen

HAL Id: hal-03233689

https://hal.sorbonne-universite.fr/hal-03233689v1

Submitted on 25 May 2021

# MATE: A Model-based Algorithm Tuning Engine

## A proof of concept towards transparent feature-dependent parameter tuning using symbolic regression

Mohamed El Yafrani[1], Marcella Scoczynski[2], Inkyung Sung[1], Markus Wagner[3], Carola Doerr[4], and Peter Nielsen[1]

[1] Operations Research group, Aalborg University, Denmark
[2] Federal University of Technology Paraná (UTFPR), Brazil
[3] Optimisation and Logistics Group, The University of Adelaide, Australia
[4] Sorbonne Université, CNRS, LIP6, Paris, France

**Abstract.** In this paper, we introduce a Model-based Algorithm Tuning Engine, namely MATE, where the parameters of an algorithm are represented as expressions of the features of a target optimisation problem. In contrast to most *static* (feature-independent) algorithm tuning engines such as *irace* and *SPOT*, our approach aims to derive the best parameter configuration of a given algorithm for a specific problem, exploiting the relationships between the algorithm parameters and the features of the problem. We formulate the problem of finding the relationships between the parameters and the problem features as a symbolic regression problem and we use genetic programming to extract these expressions in a human-readable form. For the evaluation, we apply our approach to the configuration of the (1+1) EA and RLS algorithms for the One-Max, LeadingOnes, BinValue and Jump optimisation problems, where the theoretically optimal algorithm parameters to the problems are available as functions of the features of the problems. Our study shows that the found relationships typically comply with known theoretical results – this demonstrates (1) the potential of model-based parameter tuning as an alternative to existing static algorithm tuning engines, and (2) its potential to discover relationships between algorithm performance and instance features in human-readable form.

**Keywords:** Parameter tuning · Model-based tuning · Genetic programming

## 1 Motivation

The performance of many algorithms is highly dependent on tuned parameter configurations made with regards to the user's preferences or performance criteria [4], such as the quality of the solution obtained in a given CPU cost, the smallest CPU cost to reach a given solution quality, the probability to reach a given quality, with given thresholds, and so on. This configuration task can be considered as a second layer optimisation problem [19] relevant in the fields

of optimisation, machine learning and AI in general. It is a field of study that is increasingly critical as the prevalence of the application of such methods is expanded. Over the years, a range of automatic parameter tuners have been proposed, thus leaving the configuration to a computer rather than manually searching for performance-optimised settings across a set of problem instances. These tuning environments can save time and achieve better results [2].

Among such automated algorithm configuration (AAC) tools, we cite GGA [2], ParamILS [23], SPOT [3] and irace [30]. These methods have been successfully applied to (pre-tuned) state-of-the-art solvers of various problem domains, such as mixed integer programming [21], AI planning [16], machine learning [33], or propositional satisfiability solving [24]. Figure 1 illustrates the abstract standard architecture adopted by these tools.
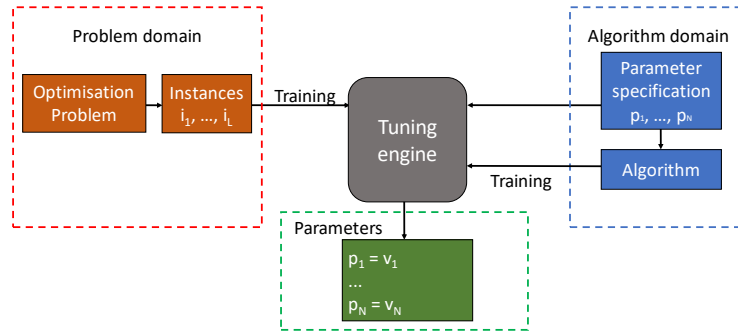


Fig. 1: Standard architecture of tuning frameworks.

However, the outcomes of these tools are static (or feature-independent), which means an algorithm configuration derived by any of these tools is not changed depending on an instance of a target optimisation problem. This leads to a significant issue as theoretical and empirical studies on various algorithms and problems have shown that parameters of an algorithm are highly dependent on features of a specific instance of a target problem [12] such as the problem size [6,35].

A possible solution to this issue is to cluster problem instances into multiple sub-groups by their size (and other potential features), then use curve fitting to map features to parameters [31,15]. A similar approach is also found in [29] that first partitions problem instances based the values of their landscape features and selects an appropriate configuration of a new problem instance based on its closeness to the partitions. However, the former approach does not scale well to multiple features and parameters, and the latter faces over-fitting issues due to the nature of the partitioning approach, making it difficult to assign an unseen instance to a specific group.

Some works have incorporated problem features in the parameter tuning process. SMAC [22] and PIAC [28] are examples of model-based tools that consider

instance features to define parameter values by applying machine learning techniques to build the model. However, an issue of these approaches is the low explainability of the outcome. For instance, while machine learning techniques such as random forest and neural networks can be used to map the parameters to problem features with a high accuracy, they are considered as black-boxes, i.e., the outcome is virtually impossible to understand or interpret. Explainability is an important concept, as not only it allows us to understand the relationships between input and output [32], but in the context of parameter tuning, it can provide an outcome that can be used to inspire fundamental research [17,18].

To tackle these issues, we propose an offline algorithm tuning approach that extracts relationships between problem features and algorithm parameters using a genetic programming algorithm framework. We will refer to this approach as MATE, which stands for Model-based Algorithm Tuning Engine. The main contributions in this work are as follows:

1. We formulate the model-based parameter tuning problem as a symbolic regression problem, where knowledge about the problem is taken into account in the form of problem features;
2. We implement an efficient Genetic Programming (GP) algorithm that configures parameters in terms of problem features; and
3. In our empirical investigation, we rediscover asymptotically-correct theoretical results for two algorithms (1+1-EA and RLS) and four problems (OneMax, LeadingOnes, BinValue, and Jump). In these experiments, MATE shows its potential in algorithm parameter configuration to produce models based on instance features.

## 2   Background

Several methods have tried to tackle the dependence between the problem features and the algorithm parameters. The Per Instance Algorithm Configuration (PIAC) [28], for example, can learn a mapping between features and best parameter configuration, building an Empirical Performance Model (EPM) that predicts the performance of the algorithm for sample (instance, algorithm/configuration) pairs. PIAC methodology has been applied to several combinatorial problems [20,36,25] and continuous domains [5].

Sequential Model-based Algorithm Configuration (SMAC) [22] is also an automated algorithm configuration tool which considers a model, usually a random forest, to design the relationship between a performance metric (e.g. the algorithm runtime) and algorithm parameter values. SMAC can also include problem features within the tuning process as a subset of input variables.

Table 1 presents a summary for some state-of-the-art methods including the approach proposed in this paper. The term 'feature-independent' means that the corresponding approach does not consider instance features. 'Model-based' approaches use a trained model (e.g. machine learning, regression, etc.) to design

Table 1: Summary of the state-of-the-art related works

| Approach Name | Algorithm | Characteristics | Ref. |
|---|---|---|---|
| **GGA** | Genetic Algorithm | Feature-independent, model-free | [2] |
| **ParamILS** | Iterated Local Search | Feature-independent, model-free | [23] |
| **irace** | Racing procedure | Feature-independent, model-free | [30] |
| **SPOT** | classical regression, tree-based, random forest and Gaussian process | Feature-independent, model-based | [3] |
| **PIAC** | Regression methods | Feature-dependent, model-based | [28] |
| **SMAC** | Random Forest | Feature-dependent, model-based | [22] |
| **MATE** | Genetic Programming | Feature-dependent, model-based, explainable | |

parameter configurations. Model-free approaches generally rely on an experimental design methodology or optimisation method to find parameter settings of an algorithm that optimise a cost metric on a given instance set.

The main differences between MATE and the other related approaches are:

1. A transparent machine learning method (GP) is utilised to enable human-readable configurations (in contrast to, e.g., random forests, neural networks, etc.).
2. The training phase is done on one specific algorithm and one specific problem in our approach – the model is less instance-focused but more problem-domain focused by abstracting via the use of features. For example, the AAC experiments behind [17,18] have guided the creation of new heavy-tailed mutation operators that were beating the state-of-the-art. Similarly, the AAC and PIAC experiments in [34] showed model dependencies on easily-deducible instance features.

Lastly, our present paper is much aligned with the recently founded research field "Data Mining Algorithms Using/Used-by Optimisers (DUO)" [1]. There, data miners can generate models explored by optimisers; and optimisers can adjust the control parameters of a data miner.

## 3    The MATE Framework

### 3.1    Problem Formulation and Notation

Let us denote an optimisation problem by $\mathcal{B}$ whose instances are characterised by the problem-specific features $\mathcal{F} = \{f_1, \ldots, f_M\}$. A target algorithm $\mathcal{A}$ with its parameters $\mathcal{P} = \{p_1, \ldots, p_N\}$ is given to address the problem $\mathcal{B}$. A set of instances $\mathcal{I} = \{i_1, \ldots, i_L\}$ of the problem $\mathcal{B}$ and a $L \times M$ matrix $\mathcal{V}$, whose element value $v_{i,j}$ represents the $j$th feature value of the $i$th problem instance, are given.

Under this setting, we define the model-based parameter tuning problem as the problem of deriving a list of mappings $\mathcal{M} = \{m_1, \ldots, m_N\}$ where each mapping $m_j \colon \mathbb{R}^M \to \mathbb{R}$, which we will refer to as a *parameter expression*, returns

a value for the parameter $p_j$ given feature values of an instance of the problem $\mathcal{B}$. Specifically, the objective of the problem is to find a parameter expression set $\mathcal{M}^*$, such that the performance of the algorithm $\mathcal{A}$ across all the problem instances in $\mathcal{I}$ is optimised.

## 3.2 Architecture Overview

In this section, we introduce our approach for parameter tuning based on the problem features. Figure 2 illustrates the architecture of the MATE tuning engine. In contrast to static methods, we consider the features of the problem. These feature are to be used in the training phase in addition to the instances, the target algorithm and the parameter specifications. Once the training is finished, the model can be used on unseen instances to find the parameters of the algorithm in terms of the problem feature values of the instance.
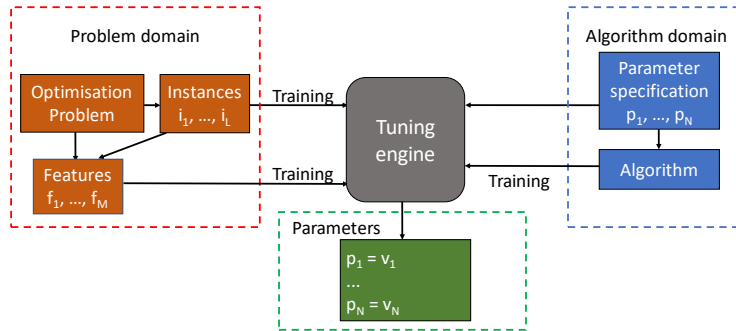


Fig. 2: Architecture of the proposed MATE framework

For example, a desired outcome of applying the MATE framework can be:

- Mutation probability of an evolutionary algorithm in terms of the problem size;
- Perturbation strength in an iterated local search algorithm in terms of the ruggedness of the instance and the problem size; and
- Population size of an evolutionary algorithm in terms of the problem size.

Note that all the examples include the problem size as a problem feature. In both theory and practice, the problem size is among the most important problem features, and it is usually known prior to the optimisation, without any need for a pre-processing step. More importantly, an extensive number of theoretical studies showed that the optimal choice of parameters is usually expressed in terms of the problem size (see, e.g. [6,12,35]).

### 3.3   The Tuning Algorithm

We use a tree-based Genetic Programming system as the tuning algorithm. It starts with a random population of trees, where each tree represents a potential parameter expression. Without loss of generality, we assume that the target problem is always a maximisation problem[5].

**The Score Function and Bias Reduction**  The score function is expressed as the weighted sum of the obtained objective values on each instance in the training set $\mathcal{I}$. Using the notations previously introduced, the score is defined in Equation (1):

$$S(t) = \frac{1}{L} \Sigma_{i \in I} \frac{z_{\mathcal{A}}(m_1(v_{i,1}, \ldots, v_{i,M}), \ldots, m_N(v_{i,1}, \ldots, v_{i,M}), i)}{R_i} \qquad (1)$$

where:

- $S(.)$ is the GP score function,
- $z_{\mathcal{A}}(\varphi_1, \ldots, \varphi_N, i)$ is a function measuring the goodness of applying the algorithm $A$ with the parameter values $\varphi_1, \ldots, \varphi_N$ to instance $i$,
- $R_i$ is the best known objective value for instance $i$.

The weights are used as a form of normalisation to reduce the bias some instances might induce. A solution to address this issue would be to use the optimal value or a tight upper bound. However, since we assume the such values are unknown (the problem itself can be unknown), we use the best known objective value ($R_i$) as a reference instead. In order to always ensure that score is well contained, the reference values are constantly updated whenever possible during the tuning process.

**Replacement Strategy – Statistical Significance and Bloat Control**  As the target algorithm can be stochastic, it is mandatory to perform multiple runs to ensure statistical significance (refer to Table 3). Thus, the replacement of trees is done based on the Wilcoxon rank-sum test.

Another aspect to take into account during the replacement process is bloat control. In our implementation, we use a simple bloat minimisation method based on the size of tree (number of nodes).

Given a newly generated tree ($Y$), we compare it against each tree ($X$) in the current population starting from the ones with the lowest scores using the following rules:

- If $Y$ is deemed to be significantly better than $X$ (using the Wilcoxon test). then we replace $X$ with $Y$ irrespective of the sizes.
- If there is no statistical significance between $X$ and $Y$, but $Y$ has a smaller size than $X$, then we replace $X$ with $Y$.
- Otherwise, we do not perform the replacement.

---

[5] The current MATE implementation is publicly available at `https://gitlab.com/yafrani/mate`

Table 2: Summary of problems

| Problem | Features | Training set |
|---|---|---|
| **OneMax**$(n)$ | $n$: number of bits | $n = 10, 20, 50, 100, 200, 500$ |
| **BinValue**$(n)$ | $n$: number of bits | $n = 10, 20, 50, 100, 200, 500$ |
| **LeadingOnes**$(n)$ | $n$: number of bits | $n = 10, 20, 50, 100, 200, 500$ |
| **Jump**$(m, n)$ | $m$: width of region with bad fitness values<br>$n$: number of bits | $(m, n) = (2, 10), (3, 10), (4, 10), (5, 10),$<br>$(2, 20), (3, 20), (4, 20),$<br>$(2, 50), (3, 50),$<br>$(2, 100), (3, 100),$<br>$(2, 200)$ |

Table 3: MATE setup

| Attribute/Parameter | Value/Content |
|---|---|
| Terminals | $\{1, 2, -1, -2\} \bigcup \mathcal{F}$ |
| Functions | Arithmetic operators |
| Number of GP generations | 100 |
| Population size | 20 |
| Tournament size | 5 |
| Replacement rate | $< 75\%$ |
| Initialisation | grow (50%) and full (50%) methods |
| Mutation operator | random mutations |
| Mutation probability | 0.2 |
| Crossover operator | sub-tree gluing |
| Crossover rate | 80% |
| Number of independent runs of target algorithm | 10 |
| $p$-value for the Wilcoxon ranksum test | 0.02 |

## 4 Computational Study

### 4.1 Experimental Setting

To evaluate our framework, we consider two target algorithms, the (1+1) EA$(\mu)$ and RLS$(k)$. The (1+1) EA$(\mu)$ is a simple hill-climber which uses standard bit mutation with mutation rate $\mu$. RLS$(k)$ differs from the (1+1) EA$(\mu)$ only in that it uses the mutation operator that always flips $k$ uniformly chosen, pairwise different bits. That is, the mutation strength $k$ is deterministic in RLS, whereas it is binomially distributed in case of the (1+1) EA$(\mu)$, Bin$(n, \mu)$, where $n$ is the number of bits.

We use MATE to configure the two algorithms for the four different problems with different time budgets as summarised in Table 2. In the table, the features of the problems used to tune the algorithm parameters and the different feature values chosen to generate problem instances of the problems are also presented. These problems have been chosen because they are among the best studied benchmark problems in the theory of evolutionary algorithms [13]. The details of our GP implementation for the experiments are presented in Table 3. Based on Table 3 and the set of features, our GP method uses a minimalistic set of 6 terminals at most: $m$, $n$ and $\{1, 2, -1, -2\}$.

It is worth noting that we are focusing in this paper on tuning algorithms with a single parameter. This is done to deliver a first prototype that is validated on

algorithms and problems extensively studied by the EA theory community. An extension to tuning several algorithm parameters forms an important direction for future work.

For example, given a budget of $(1 + o(1))en\ln(n)$, it is known that the $(1+1)$EA$(1/n)$ optimises the OneMax function as well as any other linear functions with a decent probability. It is also known that the $1/n$ is asymptotically optimal [27]. Note, though, that such *fixed-budget results* are still very sparse [26], since the theory of EA community largely focuses on expected optimisation times. Since these can nevertheless give some insight into the optimal parameter settings, we note the following:

- OneMax and BinValue: the $(1+1)$EA$(1/n)$ optimises every linear function in expected time $en\ln(n)$, and no parameter configuration has smaller expected running time, apart from possible lower order terms [35]. For RLS, it is not difficult to see that $k = 1$ yields an expected optimisation time of $(1 + o(1))n\ln(n)$, and that this is the optimal (static) mutation strength;
- LeadingOnes: on average, RLS(1) needs $n^2/2$ steps to optimise LeadingOnes. This choise also minimises the expected optimisation time. For the $(1+1)$ EA, $\mu \approx 1.59/n$ minimises the expected optimisation time, which is around $0.77n^2$ for this setting [6]. The standard mutation rate $\mu = 1/n$ requires $0.86n^2$ evaluations, on average, to locate the optimum, of the LeadingOnes function. For LeadingOnes, it is known that the optimal parameter setting drastically depends on the available budget. This can be inferred from the proofs in [6,9]; and
- Jump: mutation rate $m/n$ minimises the expected optimisation time of the $(1+1)$ EA on Jump$(m,n)$, which is nevertheless $\Theta((e/m)^m n^m)$ [12].

### 4.2   Performance Analysis

**Training Phase** The experimental study is conducted by running MATE ten times on each algorithm, problem and budget combination (refer to Table 4 for the list of budgets). This results in an elite population of 20 individuals for each setting, from which we select the top 5 expressions in terms of the score. These results are then merged and the 3 most frequent expressions are selected. For instance, the expression $2/n$ for OneMax with $0.5enln(n)$ appears 92 times over the 200 individuals (population size (20) $\times$ runs (10)).

In the current implementation, expression types (integers and non-integers) are not taken into account during the evolution. Therefore, the resulting expressions are converted into integers in the case of RLS by merging all real numbers $r$ using $\lfloor r \rfloor$ (e.g. $k = 3/2$ will be replaced by $k = 1$). On the other hand, expressions are simplified for EA by eliminating additive constants (e.g. $\mu = 1/(n+1)$ is replaced by $\mu = 1/n$).

**Evaluation Phase I** To assess the performance of MATE, we evaluate for each problem-budget combination each of the top 3 most frequent expressions,

by running them 100 independent times on each training dimension. We then normalise the outputs as in Equation (1). The results are shown in the box plots in Table 4.

*Comparison amongst the top 3 configurations.* When comparing the top 3 ranked configurations, we observe the following from Table 4 while we compare medians.

- OneMax: For (1+1) EA, $\mu = 1/n$, which ranked second for budgets $0.5en \ln n$ and $en \ln n$ and first for budget $2en \ln n$ performs better than $\mu = 1/2 * n$; while for RLS, the expression $k = 1$ appears at least on 94%, providing the best results;
- BinValue: $\mu = 1/n$ represents 18% on $en \ln n$ for (1+1) EA experiments, and a similar performance with $\mu = 2/n$ and $\mu = 3/n$; while on $0.5en \ln n$ case the $\mu = 1/n$ expression provides better results than $\mu = 1/2$ and $\mu = 1/3$; on the same way the expression $k = 1$ corresponds to 60% of the cases on RLS with the budget of $2n \ln n$ with a better performance than $k = 2$ and $k = n$;
- LeadingOnes: $\mu = 1/n$ is the most frequent expression among all considered budgets on (1+1) EA and $\mu = 2/n$ presents the best performance amongst the top 3 expressions for all budget cases; $k = 1$ represents 88% on RLS cases with $0.75n^2$ iterations and performs better than $k = 2$ and $k = 3$ for both considered budgets.
- Jump: $\mu = 2/n$ and $\mu = m/n$ present similar results for both budget cases; $\mu = 1/n$ appears on 36% and 68% of the cases on (1+1) EA on the considered budgets respectively, and performs worse than the other two $\mu$ configurations; for RLS experiments $k = m$ is the most frequent expression and performs better than $k = 2 * m$ and $k = 3$.

*Comparison of top 3 configurations against other parameter settings.* For a fair assessment of our results, we add to this comparison some expressions that were not ranked in the top 3. These are $\mu = i/n$ with $i \in \{1, 3/2, 2, 5/2, 3, 4\}$ for (1+1) EA($\mu$) for OneMax and LeadingOnes. For readability purposes, the top 3 expressions are complemented with 3 of these additional expressions in the same order they are shown. We can observe in Table 4 that these additional expressions present low frequencies, $\mu = 3/n$ being the highest case with 12% with the budget $en \ln n$, while expressions $\mu = 3/(2n)$ and $\mu = 5/(2n)$ are the lowest cases among the considered budgets. Note that the frequencies do not necessarily sum up to 100% as other expressions not reported here might have occurred.

*Comparison with theoretical results.* As we have mentioned in the beginning of this section, one should be careful when comparing theoretical results that have been derived either in terms of running time or in terms of asymptotic convergence analysis, as typically done in runtime analysis. It is well known that optimal parameter settings for concrete (typically, comparatively small) dimensions can be different from the asymptotically optimal ones [7,8]. We nevertheless see that the configurations that minimise the expected running times (again, in the classical, asymptotic sense) also show up in the top 3 ranked configurations. In Table 4, we highlight the asymptotically optimal best possible running time

Table 4: Results for 20 settings.

| | | 1+1-EA | | RLS | |
|---|---|---|---|---|---|
| | | **Budget** | **Result** | **Budget** | **Result** |
| **OneMax** | | $0.5en\ln(n)$ |  2/n [46%]<br>1/n [32%]<br>1/(2*n) [10%]<br>3/n [4%]<br>5/(2*n) [2%]<br>3/(2*n) [0%] | $n\ln(n)^*$ |  1 [98%]<br>3 [2%]<br>2 [0%] |
| | | $en\ln(n)^*$ |  2/n [46%]<br>1/n [26%]<br>1/(2*n) [14%]<br>3/n [12%]<br>5/(2*n) [2%]<br>3/(2*n) [0%] | $2n\ln(n)^{**}$ |  1 [94%]<br>3 [4%]<br>2 [2%] |
| | | $2en\ln(n)^{**}$ |  1/n [44%]<br>2/n [26%]<br>1/(2*n) [12%]<br>3/n [8%]<br>3/(2*n) [8%]<br>5/(2*n) [0%] | | |
| **BinValue** | | $0.5en\ln(n)$ |  1/2 [36%]<br>1/n [26%]<br>1/3 [6%] | $0.5n\ln(n)$ |  n [42%]<br>2 [32%]<br>1 [22%] |
| | | $en\ln(n)^*$ |  2/n [44%]<br>1/n [18%]<br>3/n [14%] | $n\ln(n)^*$ |  2 [40%]<br>n [36%]<br>1 [14%] |
| | | $2en\ln(n)^{**}$ |  2/n [48%]<br>3/n [18%]<br>1/n [12%] | $2n\ln(n)^{**}$ |  1 [60%]<br>2 [32%]<br>n [6%] |
| **LeadingOnes** | | $0.5n^2$ |  1/n [52%]<br>2/n [28%]<br>4/n [20%]<br>3/n [0%]<br>5/(2*n) [0%]<br>3/(2*n) [0%] | $0.5n^2*$ |  2 [70%]<br>1 [26%]<br>3 [4%] |
| | | $0.8n^{2**}$ |  1/n [62%]<br>2/n [18%]<br>3/n [16%]<br>4/n [2%]<br>5/(2*n) [0%]<br>3/(2*n) [0%] | $0.75n^{2**}$ |  1 [88%]<br>2 [12%]<br>3 [0%] |
| | | $0.9n^{2**}$ |  1/n [48%]<br>2/n [28%]<br>3/n [18%]<br>5/(2*n) [4%]<br>4/n [0%]<br>3/(2*n) [0%] | | |
| **Jump** | | $n^m$ |  1/n [36%]<br>2/n [32%]<br>m/n [12%] | $n^m$ |  m [34%]<br>2*m [24%]<br>3 [20%] |
| | | $en^{m**}$ |  1/n [68%]<br>2/n [22%]<br>m/n [6%] | $2n^m$ |  m [42%]<br>2*m [20%]<br>3 [18%] |

† The $y$-axis show the best found expressions with its frequency between square brackets, and the $x$-axis represents the normalised fitness.

by an asterisk *. Budgets exceeding this bound are marked by two asterisks **. As for the individual problems, we note the following:

- OneMax: It is interesting to note here that the performance is not monotonic in $k$, i.e., $k = 2$ performs worse than $k = 1$ and $k = 3$. This is caused by a phenomenon described in [11, Section 4.3.1], which states that, regardless of the starting point, the expected progress is always maximised by an uneven mutation strength. MATE correctly identifies this and suggests uneven mutation strengths in almost all cases.
- BinValue: We observe that it is very difficult here to distinguish the performance of the different configurations. This is in the nature of BinValues, as setting the first bit correctly already ensures 50% of the optimal fitness values. We very drastically see this effect in the recommendation to use $k = n$ for the RLS cases. With this configuration, the algorithm evaluates only two points: the random initial point $x$ and its pairwise complement $\bar{x}$, regardless of the budget. As can be seen in Table 4, the performance of this simple strategy is quite efficient, and hard to beat
- LeadingOnes: As mentioned earlier, for the (1+1) EA, the optimal mutation rate in terms of minimising the expected running time is around $\mu = 1.59/n$. We see that $\mu = 3/(2n)$, which did not show in the top 3 ranked configurations performs better than any of the suggestions by MATE.
- Jump: as discussed, mutation rate $\mu = m/n$ minimises the expected optimisation time. MATE recognises it as a good configuration in some of the runs. However, we see that $\mu = 2/n$, which equals $\mu = m/n$ for 5 out of our 12 training sets, shows comparable performance, and in the $en^m$ budget case even slightly better performance.

**Evaluation Phase II** To properly assess the performance of MATE, we conducted experiments for OneMax and LeadingOnes instances of larger sizes that were not considered in the training phase. The goal of this experiment is to empirically demonstrate that our approach generalises well for large and unseen instances. These results are presented in Table 5 where 100 runs were performed for OneMax with $n \in \{1000, 2000, 5000\}$ and LeadingOnes with $n \in \{750, 1000\}$. We can observe the following:

- There is less overlap amongst the confidence intervals especially for smaller budgets, which means there is a higher level of separability amongst the performances of the different expressions.
- By comparing these results with the ones from Table 4, we can observe that the results of the top 3 expressions on large instances are statistically better in the majority of cases.
- OneMax: For (1+1) EA, in contrast to the results in Table 4 where $\mu = 1/n$ and $\mu = 3/(2n)$ show a similar performance, here $\mu = 1/n$ performs better than the other expressions. For RLS, the best performing expression is $k = 1$, which was ranked first.

Table 5: Results for larger OneMax and LeadingOnes instances

| | 1+1-EA | | RLS | |
|---|---|---|---|---|
| | **Budget** | **Result** | **Budget** | **Result** |
| **OneMax** | $0.5en\ln(n)$ | 2/n [46%]<br>1/n [32%]<br>1/(2*n) [10%]<br>3/n [4%]<br>5/(2*n) [2%]<br>3/(2*n) [0%] | $n\ln(n)$* | 1 [98%]<br>3 [2%]<br>2 [0%] |
| | $en\ln(n)$* | 2/n [46%]<br>1/n [26%]<br>1/(2*n) [14%]<br>3/n [12%]<br>5/(2*n) [2%]<br>3/(2*n) [0%] | $2n\ln(n)$** | 1 [94%]<br>3 [4%]<br>2 [2%] |
| | $2en\ln(n)$** | 1/n [44%]<br>2/n [26%]<br>1/(2*n) [12%]<br>3/(2*n) [8%]<br>3/n [8%]<br>5/(2*n) [0%] | | |
| **LeadingOnes** | $0.5n^2$ | 1/n [52%]<br>2/n [28%]<br>4/n [20%]<br>3/n [0%]<br>5/(2*n) [0%]<br>3/(2*n) [0%] | $0.5n^2$* | 2 [70%]<br>1 [26%]<br>3 [4%] |
| | $0.8n^2$** | 1/n [62%]<br>2/n [18%]<br>3/n [16%]<br>4/n [2%]<br>3/(2*n) [0%]<br>5/(2*n) [0%] | $0.75n^2$** | 1 [88%]<br>2 [12%]<br>3 [0%] |
| | $0.9n^2$** | 1/n [48%]<br>2/n [28%]<br>3/n [18%]<br>5/(2*n) [4%]<br>4/n [0%]<br>3/(2*n) [0%] | | |

– LeadingOnes: For (1+1) EA the best expressions are $\mu = 2/n$, which was ranked second, and $\mu = 3/(2n)$, which was not ranked among the top 3 expressions. For RLS, $k = 1$, ranked first and second, is the best performing expression.

### 4.3   Comparative study

Herein, we compare the performance of MATE with irace and SMAC. The goal is to investigate the sensitivity of the obtained parameters on unseen instances. For a fair comparison, we run irace and SMAC with 2000 maximum experiments (which we believe is equivalent to the 100 GP generations with a population size of 20 individuals in MATE) considering the training instances presented in Table 2. We report the best elite parameter values returned by irace (2 candidates), SMAC (1 candidate) and MATE (most frequent expressions) in the columns $\mu$

and $k$ in Table 6, while the score (Eq. 1) is shown in column *Score* with the standard deviation as a subscript. These parameter values are then applied over 100 runs performed for OneMax with $n \in \{1000, 2000, 5000\}$ and LeadingOnes with $n \in \{750, 1000\}$.

Table 6: Results for MATE, irace and SMAC for OneMax and LeadingOnes instances.

| | | | **1+1-EA** | | | | | | | **RLS** | | | | |
| | | | MATE | irace | | SMAC | | | | MATE | | irace | | SMAC |
| | Budget | $\mu$ | Score | $\mu$ | Score | $\mu$ | Score | Budget | $k$ | Score | $k$ | Score | $k$ | Score |
| **OneMax** | $\frac{enln(n)}{2}$ | $\frac{2}{n}$ | $0.99_{0.001}$ | $0.258$ | $0.57_{0.002}$ | $0.009$ | $0.8_{0.003}$ | $nln(n)$ | 1 | $1_0$ | 1 | $1_0$ | 1 | $1_0$ |
| | | $\frac{1}{n}$ | $0.99_{0.001}$ | $0.216$ | $0.58_{0.002}$ | | | | 3 | $0.96_{0.002}$ | | | | |
| | | $\frac{1}{2n}$ | $0.98_{0.002}$ | | | | | | 2 | $0.94_{0.003}$ | | | | |
| | $enln(n)$ | $\frac{1}{n}$ | $1_0$ | $0.009$ | $0.82_{0.002}$ | $0.016$ | $0.76_{0.003}$ | $2nln(n)$ | 1 | $1_0$ | 1 | $1_0$ | 1 | $1_0$ |
| | | $\frac{2}{n}$ | $1_0$ | $0.013$ | $0.79_{0.003}$ | | | | 3 | $0.98_{0.001}$ | | | | |
| | | $\frac{1}{2n}$ | $1_0$ | | | | | | 2 | $0.97_{0.002}$ | | | | |
| | $2enln(n)$ | $\frac{2}{n}$ | $1_0$ | $0.594$ | $0.54_{0.002}$ | $0.008$ | $0.86_{0.002}$ | | | | | | | |
| | | $\frac{1}{n}$ | $1_0$ | $0.589$ | $0.54_{0.002}$ | | | | | | | | | |
| | | $\frac{1}{2n}$ | $1_0$ | | | | | | | | | | | |
| **LeadingOnes** | $0.5n^2$ | $\frac{1}{n}$ | $0.7_{0.025}$ | $0.430$ | $0.03_{0.002}$ | $0.024$ | $0.29_{0.007}$ | $0.5n^2$ | 2 | $0.86_{0.014}$ | 1 | $0.98_{0.026}$ | 1 | $0.98_{0.02}$ |
| | | $\frac{2}{n}$ | $0.8_{0.021}$ | $0.409$ | $0.03_{0.002}$ | | | | 1 | $0.97_{0.027}$ | 5 | $0.61_{0.01}$ | | |
| | | $\frac{4}{n}$ | $0.71_{0.015}$ | | | | | | 3 | $0.75_{0.013}$ | | | | |
| | $0.8n^2$ | $\frac{1}{n}$ | $0.95_{0.023}$ | $0.255$ | $0.05_{0.002}$ | $0.005$ | $0.83_{0.017}$ | $0.75n^2$ | 1 | $1_0$ | 1 | $1_0$ | 1 | $1_0$ |
| | | $\frac{2}{n}$ | $0.99_{0.012}$ | $0.258$ | $0.05_{0.002}$ | | | | 2 | $0.95_{0.009}$ | | | | |
| | | $\frac{3}{n}$ | $0.91_{0.018}$ | | | | | | 3 | $0.82_{0.01}$ | | | | |
| | $0.9n^2$ | $\frac{1}{n}$ | $0.99_{0.011}$ | $0.158$ | $0.07_{0.003}$ | $0.006$ | $0.75_{0.013}$ | | | | | | | |
| | | $\frac{2}{n}$ | $1_0$ | $0.153$ | $0.07_{0.006}$ | | | | | | | | | |
| | | $\frac{3}{n}$ | $0.95_{0.014}$ | | | | | | | | | | | |

Table 6 shows that MATE significantly outperforms irace and SMAC for (1+1) EA. On the other hand, the three methods show a similar performance on RLS. This is due to the fact that the parameter $\mu$ in (1+1) EA is highly sensitive to the problem feature $n$. In contrast, the parameter $k$ in RLS is independent from $n$ and its best value ($k = 1$) was identified by the three methods for both OneMax and LeadingOnes.

## 5    Conclusions and Future Directions

With this article, we have presented MATE as a model-based algorithm tuning engine: its human-readable models map instance features to algorithm parame-

ters. Our experiments showed that MATE can find known asymptotic relationships between the feature values and algorithm parameters. We also compared the performance of MATE with iRace and SMAC investigating the sensitivity of the obtained parameters on unseen instances of larger size. With its scalable models, MATE performed best. It is worth noting that MATE can be a useful guideline tool for theory researchers due to its white-box nature, similarly to how results in [14] inspired the analysis of a generalised one-fifth success rule in [10]. But MATE can also be extended to be used as a practical toolbox for feature-based algorithm configuration.

In the future, we intend to explore, among other, the following three avenues. First, the design of MATE itself will be subject to extensions, e.g. to better handle performance differences between instances via ranks or racing. Second, while our proof-of-concept study here was motivated by theoretical insights, we will investigate more realistic problems for which instance features are readily available, such as the travelling salesperson problem and the assignment problem. Third, we will investigate approaches to extend MATE to handle multiple parameters to demonstrate its ability to tune more sophisticated algorithms.

## Acknowledgements

## References

1. Agrawal, A., Menzies, T., Minku, L.L., Wagner, M., Yu, Z.: Better software analytics via "duo": Data mining algorithms using/used-by optimizers. Empirical Software Engineering **25**(3), 2099–2136 (2020)
2. Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: International Conference on Principles and Practice of Constraint Programming, CP'09. pp. 142–157. Springer (2009)
3. Bartz-Beielstein, T., Flasch, O., Koch, P., Konen, W., et al.: SPOT: A toolbox for interactive and automatic tuning in the R environment. In: Proceedings. vol. 20, pp. 264–273 (2010)
4. Belkhir, N., Dréo, J., Savéant, P., Schoenauer, M.: Feature based algorithm configuration: A case study with differential evolution. In: Parallel Problem Solving from Nature, PPSN'16. pp. 156–166. Springer (2016)
5. Belkhir, N., Dréo, J., Savéant, P., Schoenauer, M.: Per Instance Algorithm Configuration of CMA-ES with Limited Budget. In: Genetic and Evolutionary Computation Conference. pp. 681–688. GECCO '17, ACM (2017)
6. Böttcher, S., Doerr, B., Neumann, F.: Optimal fixed and adaptive mutation rates for the LeadingOnes problem. In: Parallel Problem Solving from Nature, PPSN'10. LNCS, vol. 6238, pp. 1–10. Springer (2010)

7. Buskulic, N., Doerr, C.: Maximizing drift is not optimal for solving onemax. In: Genetic and Evolutionary Computation Conference, GECCO'19. pp. 425–426. ACM (2019), full version available at `http://arxiv.org/abs/1904.07818`

8. Chicano, F., Sutton, A.M., Whitley, L.D., Alba, E.: Fitness probability distribution of bit-flip mutation. Evolutionary Computation **23**(2), 217–248 (2015)

9. Doerr, B.: Analyzing randomized search heuristics via stochastic domination. Theor. Comput. Sci. **773**, 115–137 (2019)

10. Doerr, B., Doerr, C., Lengler, J.: Self-adjusting mutation rates with provably optimal success rules. In: Proc. of Genetic and Evolutionary Computation Conference (GECCO'19). pp. 1479–1487. ACM (2019). https://doi.org/10.1145/3321707.3321733, `https://doi.org/10.1145/3321707.3321733`, full version available at `https://arxiv.org/abs/1902.02588`

11. Doerr, B., Doerr, C., Yang, J.: Optimal parameter choices via precise black-box analysis. Theor. Comput. Sci. **801**, 1–34 (2020)

12. Doerr, B., Le, H.P., Makhmara, R., Nguyen, T.D.: Fast genetic algorithms. In: Genetic and Evolutionary Computation Conference, GECCO'17. pp. 777–784. ACM (2017)

13. Doerr, B., Neumann, F.: Theory of Evolutionary Computation—Recent Developments in Discrete Optimization. Springer (2020)

14. Doerr, C., Wagner, M.: Simple on-the-fly parameter selection mechanisms for two classical discrete black-box optimization benchmark problems. In: Proc. of Genetic and Evolutionary Computation Conference (GECCO'18). pp. 943–950. ACM (2018). https://doi.org/10.1145/3205455.3205560, `https://doi.org/10.1145/3205455.3205560`

15. El Yafrani, M., Ahiod, B.: Efficiently solving the traveling thief problem using hill climbing and simulated annealing. Information Sciences **432**, 231–244 (2018)

16. Fawcett, C., Helmert, M., Hoos, H., Karpas, E., Röger, G., Seipp, J.: Fd-autotune: Domain-specific configuration using fast downward. In: ICAPS 2011 Workshop on Planning and Learning. pp. 13–17 (2011)

17. Friedrich, T., Göbel, A., Quinzan, F., Wagner, M.: Heavy-tailed mutation operators in single-objective combinatorial optimization. In: Parallel Problem Solving from Nature, PPSN XV. pp. 134–145. Springer (2018)

18. Friedrich, T., Quinzan, F., Wagner, M.: Escaping large deceptive basins of attraction with heavy-tailed mutation operators. In: Genetic and Evolutionary Computation Conference. p. 293–300. GECCO '18, ACM (2018)

19. Hoos, H.H.: Programming by optimization. Commun. ACM **55**(2), 70–80 (2012)

20. Hutter, F., Hamadi, Y., Hoos, H.H., Leyton-Brown, K.: Performance prediction and automated tuning of randomized and parametric algorithms. In: International Conference on Principles and Practice of Constraint Programming, CP'06. pp. 213–228. Springer (2006)

21. 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)

22. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: International Conference on Learning and Intelligent Optimization, LION'11. pp. 507–523. Springer (2011)

23. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: Paramils: an automatic algorithm configuration framework. Journal of Artificial Intelligence Research **36**, 267–306 (2009)

24. Hutter, F., Lindauer, M., Balint, A., Bayless, S., Hoos, H., Leyton-Brown, K.: The configurable SAT solver challenge (CSSC). Artificial Intelligence **243**, 1–25 (2017)
25. Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K.: Algorithm runtime prediction: Methods & evaluation. Artificial Intelligence **206**, 79–111 (2014)
26. Jansen, T.: Analysing stochastic search heuristics operating on a fixed budget. In: Theory of Evolutionary Computation: Recent Developments in Discrete Optimization, pp. 249–270. Springer (2020)
27. Lengler, J., Spooner, N.: Fixed budget performance of the (1+1) EA on linear functions. In: ACM Conference on Foundations of Genetic Algorithms, FOGA'15. pp. 52–61. ACM (2015)
28. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In: International Conference on Principles and Practice of Constraint Programming. pp. 556–572. Springer (2002)
29. Liefooghe, A., Derbel, B., Verel, S., Aguirre, H., Tanaka, K.: Towards landscape-aware automatic algorithm configuration: preliminary experiments on neutral and rugged landscapes. In: European Conference on Evolutionary Computation in Combinatorial Optimization, EvoCOP'17. pp. 215–232. Springer (2017)
30. 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)
31. Mascia, F., Birattari, M., Stützle, T.: Tuning algorithms for tackling large instances: an experimental protocol. In: International Conference on Learning and Intelligent Optimization. pp. 410–422. Springer (2013)
32. Rai, A.: Explainable ai: from black box to glass box. Journal of the Academy of Marketing Science **48**(1), 137–141 (2020)
33. Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. In: Advances in neural information processing systems. pp. 2951–2959 (2012)
34. Treude, C., Wagner, M.: Predicting good configurations for github and stack overflow topic models. In: 16th International Conference on Mining Software Repositories. p. 84–95. MSR '19, IEEE (2019)
35. Witt, C.: Tight bounds on the optimization time of a randomized search heuristic on linear functions. Combinatorics, Probability & Computing **22**, 294–318 (2013)
36. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research **32**, 565–606 (2008)