Pool-Based Realtime Algorithm Configuration: A Preselection Bandit Approach

Adil El Mesaoudi-Paul[1], Dimitri Weiß[2], Viktor Bengs[1(✉)], Eyke Hüllermeier[1], and Kevin Tierney[2]

[1] Heinz Nixdorf Institute and Department of Computer Science, Paderborn University, Paderborn, Germany
viktor.bengs@upb.de

[2] Decision and Operation Technologies Group, Bielefeld University, Bielefeld, Germany
{dimitri.weiss,kevin.tierney}@uni-bielefeld.de

Abstract. The goal of automatic algorithm configuration is to recommend good parameter settings for an algorithm or solver on a per-instance basis, i.e., for the specific problem instance being solved. Realtime algorithm configuration is a practically motivated variant of algorithm configuration, in which the problem instances arrive in a sequential manner and high-quality configurations must be chosen during runtime. We model the realtime algorithm configuration problem as an extended version of the recently introduced contextual preselection bandit problem. Our approach combines a method for selecting configurations from a pool of candidates with a surrogate configuration generation procedure based on a genetic crossover procedure. In contrast to existing methods for realtime algorithm configuration, the approach based on contextual preselection bandits allows for the incorporation of problem instance features as well as parameterizations of algorithms. We test our algorithm on different realtime algorithm configuration scenarios and find that it outperforms the state of the art.

Keywords: Realtime algorithm configuration · Contextual preselection bandit

1 Introduction

It is widely known that no single solver produces optimal performance for all types of problem instances [11]. Due to the large space of parameterizations available for most solvers, algorithm designers are forced to tune the parameters of their approaches to provide reasonable performance, a long and arduous process. Recently, automatic algorithm configuration has simplified the search for good parameters by *automatically* identifying and recommending high-quality parameters to algorithm designers and users. Furthermore, these approaches can work in an *instance specific* fashion, providing high-quality parameters specific to the

I. S. Kotsireas and P. M. Pardalos (Eds.): LION 14 2020, LNCS 12096, pp. 216–232, 2020.
https://doi.org/10.1007/978-3-030-53552-0_22

instances being solved [31] including for instances never before seen or envisioned by the algorithm designer.

Traditionally, algorithm configuration research has focused on offline, train-once approaches [2,23,24]. These are based on a static set of instances deemed to be representative of a specific task for which good parameter settings are found and after which the parameters are put into practice. Ignoring the fact that a representative set of instances may not be available for a given problem at the time of algorithm design, train-once methods suffer from changes or *drift* of problem instances. Furthermore, there may be a lack of time for repeated offline training. Online or *realtime* approaches, such as the ReACT and ReACTR systems of [17] and [16], respectively, overcome these issues by choosing a high-quality configuration of the solver during runtime as new instances arrive.

In this paper, we propose an algorithm called Contextual Preselection with Plackett-Luce (CPPL) for realtime algorithm configuration. The latter is based on the contextual preselection bandit problem, a variant of the preference-based multi-armed bandit (MAB) problem [12] recently introduced in [6]. MAB methods have proven successful for the closely related problem of algorithm selection, see e.g. [14,19,22,33,38], but have not yet been applied to algorithm configuration.

The state-of-the-art realtime algorithm configurator ReACTR [16] utilizes the racing principle in which, for a given instance, multiple parameter configurations are run in parallel on multiple CPU cores to see which is best for a given instance. ReACTR gathers information about the performance of parameterizations and applies a ranking mechanism called TrueSkill [21] to decide which choice of parameterizations to select from a pool of options for the next instance. Furthermore, it uses TrueSkill to decide which parameterizations to replace with new ones and how to generate new parameterizations, meaning which individuals to choose as parents for the crossover mechanism included in the generation of new parameterizations. Our goal in this paper is to replace the ranking, choice and generation mechanisms of ReACTR by functions of the CPPL approach while maintaining the framework of the racing principle and parallel execution of ReACTR. Our contributions are as follows:

1. We connect the online contextual preselection bandit setting to the realtime algorithm configuration problem.
2. We introduce the CPPL algorithm for solving realtime algorithm configuration tasks.
3. We provide a novel technique for generating new parameterizations using the surrogate of the CPPL algorithm inspired by the idea of genetic engineering in [1].
4. We show experimentally that CPPL is competitive with the state-of-the-art ReACTR algorithm on different algorithm configuration scenarios.

The paper is organized as follows. In the next section, we give an overview of related work. The online contextual preselection setting and its application to realtime algorithm configuration are presented in Sect. 3. An experimental study is then presented in Sect. 4, prior to concluding the paper in Sect. 5.

2 Related Work

We now provide an overview of the related work in several fields: algorithm configuration, algorithm selection using MABs, and hyperparameter optimization for machine learning algorithms.

2.1 Algorithm Configuration

Approaches for offline algorithm configuration can be broadly categorized into non-model-based, model-based and hybrid approaches. Non-model-based techniques including racing algorithms such as F-Race [9] and its successor Iterated F-Race [10]. In these approaches, parameterizations are run against each other on subsets of instances and those that are performing poorly are eliminated through statistical testing. In the gender-based genetic algorithm (GGA) of [2] and its extension [1] a racing mechanism is also used to run instances in parallel and cut off poor performing parameterizations before they use too much CPU time (in a runtime minimization setting). The ParamILS method [24] employs a focused iterated local search. Sequential Model-based Algorithm Configuration (SMAC) approach by [23] is a model-based method that generates a response surface over parameters of the solver to predict the performance if those parameters were to be used. Finally, [1] can be considered a hybrid approach for configuration, utilizing both a genetic algorithm and a learned forest to select new parameterizations.

Realtime methods include two approaches. The previously mentioned ReACT algorithm [17] stores a set of parameterizations and runs them in parallel on the instances to be solved. Parameterizations are removed if they do not "win" enough of the races with the other parameterizations and are replaced with random parameterizations. ReACT's extension ReACTR [16] enhances ReACT by incorporating the ranking technique of the ranking system TrueSkill and uses a crossover mechanism (as in genetic algorithms) to generate new parameterizations. Both [17] and [16] show that tackling the realtime configuration problem head-on instead of periodically performing a new offline configuration results in better performance.

A recent contribution from [7] introduces the concept of dynamic algorithm configuration, in which the goal is to dynamically configure the parameters of an algorithm while it is running. The problem is modeled as a contextual Markov decision process and reinforcement learning is applied. Algorithm configuration has been examined from a theoretical perspective in several recent works, e.g. [5, 28] resulting in bounds on the quality of configuration procedures.

2.2 Algorithm Selection and Hyperparameter Optimization with Bandits

The K-armed bandit problem is a sequential decision problem where in each trial a learner needs to choose/pull one of the arms of a K-armed slot machine (*bandit*), with each arm having its own reward distribution. After pulling an

arm the learner observes its associated reward. The goal of the learner is to maximize the sum of rewards or, equivalently, minimize the total loss defined as the expected difference between the sum of rewards associated with the optimal selection strategy with the best possible reward and the sum of the collected rewards. The main issue affecting the long-term success of a learner in such problems is to find a good trade-off between choosing arms that have turned out to be appealing in the past (exploitation) and trying other, possibly even better arms, but whose appeal is not precisely known so far (exploration).

The problem of algorithm selection can be represented as a bandit problem in a straightforward fashion. The arms are the algorithms, and pulling an arm corresponds to running the associated algorithm on the current problem instance and observing its runtime. The objective is to minimize the total solution time over all problem instances.

Bandit approaches have a long history of success for algorithm selection. In the following, we list some of the works following this approach for different applications. In [40] the authors model the adaptive time warp control problem in the single-agent setting as a MAB problem. A MAB framework for managing and adapting a portfolio of acquisition functions in the context of Bayesian optimization with Gaussian processes is introduced in [22]. In [33] the authors model the problem of scheduling of searches with different heuristics as a MAB problem and use dynamic Thompson sampling (DTS) to solve it. Finally, in [4] a MAB-based learning technique is used to automatically select among several levels of propagation during search for adaptive constraint propagation.

Algorithm configuration is also related to hyperparameter optimization, in which the goal is to find a configuration of hyperparameters that optimize some evaluation criterion based on a given set of hyperparameters associated with some supervised learning task and a search space over them [27]. In hyperparameter optimization, the algorithm is generally a machine learning method rather than an optimization technique. The key difference to algorithm configuration is that hyperparameter optimization generally ignores algorithm runtimes and focuses on improving the quality (i.e., the output) of an approach. Nevertheless, a problem of hyperparameter optimization can be modeled as a bandit problem in the same way.

A number of MAB approaches have also been applied for hyperparameter optimization by casting it into a pure exploration nonstochastic infinite-armed bandit problem. In this way, several well-known bandit algorithms can be leveraged to the hyperparameter optimization problem such as successive halving [27,29], upper confidence bound [39] or Thompson sampling [37].

All the MAB approaches mentioned above use quantitative feedback in the form of absolute numerical rewards (e.g., runtime of a solver or accuracy of a learning method). In contrast, we use a preference-based variant of the MAB problem, in which the feedback is purely qualitative. We merely observe which algorithm performed best among a subset of algorithms on a given problem instance. This setting is a generalization of the dueling bandits problem introduced in [41].

3 Contextual Preselection for Realtime Algorithm Configuration

In this section, we present details of our approach to realtime algorithm configuration. After a brief introduction to this problem in Sect. 3.1, we recall the (contextual) preselection problem with winner feedback in Sect. 3.2. In Sect. 3.3, we then explain how realtime algorithm configuration can be cast as a problem of that kind.

3.1 Realtime Algorithm Configuration

Formally, offline algorithm configuration can be defined as follows [24]. We are given a set of problem instances $\Pi \subseteq \hat{\Pi}$ for which we want to find a good parameter setting λ from a parameter space Λ for an algorithm $\mathcal{A}$. We can evaluate the runtime or quality of some set of parameters using a performance metric $m : \Pi \times \Lambda \to \mathbb{R}$. Without loss of generality, the goal is to find some λ using the instances Π such that $\sum_{\pi \in \hat{\Pi}} m(\pi, \lambda)$ is minimal, i.e., a parameterization λ that performs well on average (across a set of problem instances).

Notice that in offline algorithm configuration we are provided a large set of instances up front and only test the quality of the parameter configuration we find in an offline setting. In contrast, *realtime* algorithm configuration considers Π to be a *sequence* of instances that are solved one after the other. Furthermore, we assume that the underlying distribution of instances $\hat{\Pi}$ is not fixed and may change over time, meaning that the parameters tuned offline may no longer be effective. This setting is a realistic one, for example for logistics companies that must solve routing problems on a daily basis for changing customers or manufacturing companies that have new sets of orders each day. Furthermore, as companies grow, shrink or adjust their business model, $\hat{\Pi}$ could drastically change and require updating.

Figure 1 shows the realtime algorithm configuration process for our approach, and those in the literature. Given a pool of parameterizations λ_1 through λ_n, we must select a limited number according to our parallel computing resources. We run the first instance on the parameterizations and, in the runtime setting, see which one finishes first. The other parameterizations are immediately stopped. We note that in ReACT and ReACTR no information is gained from these runs, i.e., the data we receive is censored. The parameterization pool and a model of how to select the parameterizations is updated and the process is repeated for the next instances.

In this paper, we make use of a recently introduced variant of the multi-armed bandit problem, called the preselection bandit problem [6], which is able to exploit (censored) "winner feedback" as described above, i.e., information about which parameterization among a finite set of parameterizations solved a problem first. Moreover, making use of a *contextualized* extension of preselection bandits [15], we are able to recommend parameterizations of a solver on a *per-instance* basis, i.e., parameterizations that do not only perform well on average but are specifically tailored to the concrete problem instance at hand.

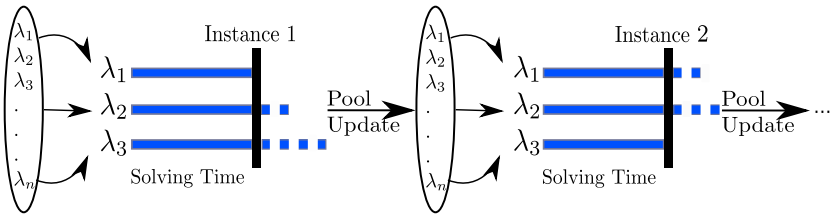


Fig. 1. Realtime algorithm configuration using the racing principle.

3.2 The Contextual Preselection Problem with Winner Feedback

The online contextual preselection problem with winner feedback, as introduced in [15], is a sequential online decision problem, in which, in each time step $t \in \{1, 2, \ldots\}$, a learner is presented a context $\mathbf{X}_t = (\boldsymbol{x}_{t,1} \ldots \boldsymbol{x}_{t,n})$. Each $\boldsymbol{x}_{t,i} \in \mathbb{R}^d$ is associated with one of n different arms, which we shall identify by $\{1, \ldots, n\}$, and contains properties of the arm itself as well as the context in which the arm needs to be chosen. After the context is revealed, the learner selects a subset $S_t \subseteq [n] = \{1, \ldots, n\}$ of $k < n$ distinct arms. Then, it obtains as feedback the top-ranked arm/winner among these arms. In each time step t, the goal of the learner is to select a subset S_t of arms that contains the best possible arm for the current context $\mathbf{X}_t$.

We consider the contextual preselection problem under the (contextualized) Plackett-Luce (PL) model [30,34], a parametrized probability distribution on the set of all rankings over a finite set of n choice alternatives. The PL model is defined by a parameter $\boldsymbol{v} = (v_1, \ldots, v_n)^\top \in \mathbb{R}_+^n$, where each v_i represents the weight or "strength" of the i-th alternative. The probability of a ranking $\boldsymbol{r}$ of the choice alternatives under the PL model is given by

$$\mathbb{P}(\boldsymbol{r} \,|\, \boldsymbol{v}) = \prod_{i=1}^{n} \frac{v_{\boldsymbol{r}^{-1}(i)}}{\sum_{j=i}^{n} v_{\boldsymbol{r}^{-1}(j)}}.$$

Here, a ranking $\boldsymbol{r}$ is a bijective mapping $\boldsymbol{r} : [n] \to [n]$, with $\boldsymbol{r}(k)$ the rank of the k^{th} alternative and $\boldsymbol{r}^{-1}(i)$ the index of the alternative on position i.

The probability that alternative k gets top-ranked among the alternatives in a subset $S \subset [n]$ under the PL model is

$$\mathbb{P}(\boldsymbol{r}_S(k) = 1 \,|\, \boldsymbol{v}) = \frac{v_k}{\sum_{i \in S} v_i}. \tag{1}$$

In order to integrate context information $\boldsymbol{x}_i \in \mathbb{R}^d$ about the i^{th} choice alternative, the constant latent utility v_i can be replaced by a log-linear function of the arm's context, in a similar way as has been done in [13,36]:

$$v_i = v_i(\mathbf{X}) = \exp\left(\theta^\top \boldsymbol{x}_i\right), \quad i \in \{1, \ldots, n\}, \tag{2}$$

where we summarize the corresponding feature vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ in a matrix $\mathbf{X} \in \mathbb{R}^{d \times n}$. The PL model with context information is then defined as

$$\mathbb{P}(\boldsymbol{r} \mid \theta, \mathbf{X}) = \prod_{i=1}^{n} \frac{v_{\boldsymbol{r}^{-1}(i)}(\mathbf{X})}{v_{\boldsymbol{r}^{-1}(i)}(\mathbf{X}) + \cdots + v_{\boldsymbol{r}^{-1}(n)}(\mathbf{X})} = \prod_{i=1}^{n} \frac{\exp\left(\theta^\top \boldsymbol{x}_{\boldsymbol{r}^{-1}(j)}\right)}{\sum_{j=i}^{n} \exp\left(\theta^\top \boldsymbol{x}_{\boldsymbol{r}^{-1}(j)}\right)}.$$

The probability for an alternative $k \in S$ to get the top rank among the alternatives in S is analogous to (1) and in particular, the corresponding log-likelihood function for an observation $(k, S, \mathbf{X})$ is

$$\mathcal{L}(\theta \mid k, S, \mathbf{X}) = \theta^\top \boldsymbol{x}_k - \log\left(\sum_{j \in S} \exp\left(\theta^\top \boldsymbol{x}_j\right)\right) \quad . \tag{3}$$

The gradient and Hessian matrix of the log-likelihood function can be computed easily and are used for deriving confidence bounds $c_{t,i}$ on the contextualized utility parameters v_i as specified in (2). The confidence bounds can be written as $c_{t,i} = \omega \cdot I(t, d, \boldsymbol{x}_{t,i})$, where $\omega > 0$ is some suitable constant and I is a function depending on the gradient as well as the Hessian of the log-likelihood function with respect to the observation at time step t. For technical details, we refer to [15].

Tackling the online contextual preselection problem requires estimating the unknown parameter vector θ and solving the well-known exploration-exploitation trade-off. In [15], the former issue is dealt with using the averaged stochastic gradient descent (SGD) method, and the latter is handled by a variant of the upper confidence bounds (UCB) algorithm. Again, we refer to [15] for a detailed description of the CPPL algorithm for the online contextual preselection problem with winner feedback and explain its adaptation to the realtime algorithm configuration problem in the next section.

3.3 Realtime Algorithm Configuration as Contextual Preselection

We model the realtime algorithm configuration problem as an online contextual preselection problem, where each algorithm parameterization is viewed as an arm, and adapt the CPPL algorithm to solve it. We note here that, because of the parametric form of the model in (2), the number of arms (parameterizations) does not need to be finite, as is normally the case in realtime algorithm configuration.

Our algorithm, called CPPL, is described in Algorithm 1. In the following, we denote the features of a parameterization $\lambda \in \Lambda$ resp. a problem instance $\pi \in \Pi$ by $f(\lambda)$ resp. $f(\pi)$. Note that for the considered problem scenario both the features of the solvers as well as the problem instances are of high dimension and reveal high correlations, so that a principal component analysis (PCA) procedure is conducted to reduce the dimensionality of the features as well as their correlation. Also note that we perform the PCA only on a small portion of the instances (20%) and the initial pool of parameterizations, under the assumption

that in a real-world setting, at least some data will be available in advance of starting the system. The obtained PCA transformations are then used to transform every new instance and parameterization features during the run of the algorithm. In the experimental study we analyze different parameter settings for the PCA to determine whether this reduces the performance of our approach or not.

The algorithm is initialized with a random parameter vector $\hat{\theta}_0$ and a random set P of n different parameterizations (line 1), which corresponds to the pool of candidates. The algorithm proceeds in discrete time steps $t = 1, 2, \ldots$. In each step, first a new problem instance $\pi \in \Pi$ is observed (line 5) and then a joint feature map of the instance features $f(\pi)$ and each of the parameterizations $f(\lambda_i)$, $i \in \{1, \ldots, n\}$ is built.

The joint feature map we use is a polynomial of the second degree, which consists of all polynomial combinations of the features with degree less than or equal to 2, i.e., it is defined for two vectors $\boldsymbol{x} \in \mathbb{R}^r$ and $\boldsymbol{y} \in \mathbb{R}^c$ as:

$$\Phi(\boldsymbol{x}, \boldsymbol{y}) = \left(1, x_1, \cdots, x_r, y_1, \cdots, y_c, x_1^2, \cdots, x_r^2, y_1^2, \cdots, y_c^2, x_1 y_1, x_1 y_2, \cdots, x_r y_c\right).$$

With this, the estimated skill parameters (see (2)) are computed by using the current estimate of θ and the joint feature map of the parameterization and problem features (line 7). Following the optimism in the face of uncertainty principle, the k most optimistic parameterizations within the pool of candidates are determined by computing the upper confidence bounds on their estimated skill parameter for the given problem instance (line 9). Generally, k corresponds to the number of the available CPU cores in the machine. These k parameterizations are then used (in parallel) to solve the given problem instance resulting in a winner feedback, i.e., the parameterization among the k used which solved the problem instance first (line 10). This winner feedback information is then used in the subsequent step to update the estimate of θ, following a stochastic gradient descent scheme (line 11). After that, poorly performing parameterizations are discarded from the pool of candidates (lines 12–13). To this end, a racing strategy is adopted [32], in which a parameterization is pruned as soon its upper bound on the estimated skill parameter is below the respective lower bound of another parameterization. Note that this differs from the preference model approach based on TrueSkill used in [16]. Finally, the parameterizations discarded in the last step are replaced by generating new ones according to a genetic approach as described in the following (line 14).

Due to the nature of the CPPL model, it is not possible to directly generate parameterizations from the learned model. Nonetheless, we wish to use the learned model to augment our candidate pool with parameterizations that will be effective on future instances. Thus, we implement a crossover-based approach based on the idea of genetic engineering [1]. We generate individuals/parameterizations using a uniform crossover operator on two individuals ranked as the best by the model. To ensure enough diversification of the search for good parameterizations, we introduce mutation of single genes as well as random generation of individuals with a lower probability. All the newly generated parameterizations are then ranked by the learned model and the best

Table 1. Comparison of CPPL and ReACTR regarding the three most important components of the realtime algorithm configuration.

	Discarding parameterizations	Generating parameterizations	Selecting for Runs
CPPL	Racing strategy	Crossover based on model and randomness, evaluation by model	Evaluation by the model
ReACTR	Based on TrueSkill	Crossover based on TrueSkill and randomness	Based on TrueSkill and random choices

individuals are selected. The parameterizations chosen to be terminated by the model are then replaced by the best parameterizations generated. Differences in approaches of solving the three main components of realtime algorithm configuration between CPPL and ReACTR are summarized in Table 1.

Algorithm 1. CPPL$(n, k, \Pi, \alpha, \gamma, \omega, f)$

Initialize n random parameterizations $P = \{\lambda_1, \ldots, \lambda_n\} \subset \Lambda$
Initialize $\hat{\theta}_0$ randomly
$\bar{\theta}_0 = \hat{\theta}_0$
for $t = 1, 2, \ldots$ **do**
  Observe problem instance $\pi \in \Pi$
  for $j = 1, \ldots, n$ **do**
    $\hat{v}_{t,j} = \exp\left(\boldsymbol{x}_{t,j}^\top \bar{\theta}_t\right)$, where $\boldsymbol{x}_{t,j} = \Phi(f(\lambda_j), f(\pi))$
  end for
  Choose S_t as:

$$\underset{S_t \subseteq [n],\ |S_t| = k}{\operatorname{argmax}} \sum_{i \in S_t} \left(\hat{v}_{t,i} + c_{t,i}\right)$$

  Run the parameterizations of S_t to solve π and observe the parameterization $w_t \in S_t$ terminating first
  Update $\bar{\theta}_t$ by $\bar{\theta}_t = (t-1)\frac{\bar{\theta}_{t-1}}{t} + \frac{\hat{\theta}_t}{t}$ with $\hat{\theta}_t = \hat{\theta}_{t-1} + \gamma t^{-\alpha} \nabla \mathcal{L}\left(\hat{\theta}_{t-1} | w_t, S_t, \mathbf{X}_t\right)$
  Let: $K \leftarrow \{\lambda_i \in P \mid \exists \lambda_j \neq \lambda_i \text{ s.t. } \hat{v}_{t,j} - c_{t,j} \geq \hat{v}_{t,i} + c_{t,i}\}$
  $\Lambda \leftarrow \Lambda \setminus K$
  Generate $|\Lambda| - |K|$ new parameterizations using the genetic approach as described.
end for

4 Computational Results

In this section, we study the following research questions:

- **RQ1:** What effect does the PCA dimension have on the performance of CPPL?
- **RQ2:** What effect does the parameter ω have on the performance of CPPL?
- **RQ3:** How good is the choice provided by CPPL?
- **RQ4:** How does CPPL compare to ReACTR?

4.1 Datasets and Solvers

We first define the datasets and solvers used in our experiments. We consider three solvers: CPLEX [26], CaDiCaL [8] and Glucose [3]. CPLEX is a mixed-integer programming solver, and CaDiCaL and Glucose are satisfiability (SAT) solvers. The type of parameters of each solver are given in Table 2. We excluded categorical parameters from the configuration by CPPL, since a PCA is employed as part of this method, which is not designed for handling non-numerical variables. For ReACTR, which is not restricted in this regard, we do include all parameters in the configuration.

We choose problem instances to emulate industrial problems with drift. For CPLEX, we use the frequency assignment problem generated by a slightly altered approach from [35]. This dataset contains 1,000 problem instances which are generated by setting the number of cells to 5 and the variance of channel requirements per cell to 1.5. The necessary distance between channels is drawn from a normal distribution, and the mean requirement of channels per cell goes from 8 to 18 in 10 stages. To introduce drift into the data, we change the generation parameters after every 10 instances.

For the SAT solvers, we use two datasets. The first dataset contains 1,000 instances generated with the modularity-based random SAT instance generator [20] by setting it to make instances with 10,000 variables, 60,000 clauses, 4 literals per clause, 600 communities and we vary the modularity factor from 0.4 to 0.35 in 10 stages. The second set of 1,000 instances is generated with the power-law random SAT generator [18]. We make instances with 10,000 variables, 93,000 clauses, 4 literals per clause, 18 as the power-law exponent of variables and the power-law exponent of clauses changing as described before from 12.5 to 2.5. The instance features used are based on [25] for MIP and SAT instances. We choose 32 features for MIP and 54 features for SAT instances.

Table 2. Types of parameters being configured in each solver.

Solver	Real	Categorical	Binary
CPLEX	35	54	6
CaDiCaL	64	29	63
Glucose	15	10	92

4.2 RQ1: Choosing the PCA Dimensions

The goal of this experiment is to determine whether PCA is suitable for reducing the dimensionality of the instance and parameter feature input and determine which dimension to use for our problem settings. To this end, we run different experiments with different values for both parameters on 200 modularity-based SAT instances using the glucose solver. The results are given in Table 3, where it can be seen that a value of 3 for the PCA dimension of instance features and 5 for the PCA dimension of algorithm features lead to the best result. The key takeaway from this experiment is that changing the dimensionality of the PCA does not significantly harm CPPL's performance. We thus use these values in further experiments.

Table 3. Overall runtime of glucose in seconds for solving 200 modularity-based SAT instances using different values for the number of PCA dimension of instance features and algorithm features. The results are averaged over 3 repetitions.

# dim. of parameters	# dim. of features				
	3	5	8	10	12
3	1863.17	1861.63	1860.85	1862.08	1865.77
5	1852.27	1864.28	1866.78	1866.87	1865.32
8	1869.94	1870.34	1866.96	1865.35	1865.19
10	1869.14	1865.32	1862.67	1867.56	1866.22
12	1858.52	1863.92	1861.75	1866.75	1866.13

4.3 RQ2: Choosing ω

We again run similar experiments for glucose to determine good values of the parameter ω, which helps determine the confidence intervals $c_{t,i}$. We note here that a smaller value of ω tightens the confidence bounds of contextualized skill parameters, which in turn leads to more parameterizations being discarded, and vice versa for larger values of ω (see line 12 in Algorithm 1). Figure 2 shows the results of several values of ω. Notice that low and high values of ω only have a very slight effect on the performance. However, with $\omega = 0.001$, we get the best cumulative runtime.

Note, that we conducted similar experiments as in Sect. 4.2 for the parameters α and γ where we found the best performance for the values of 0.2 for α and 1 for γ.

4.4 RQ3: Evaluation of the CPPL Choice

We now compare the performance of the CPPL choice with the choice of ReACTR. For this we use the glucose solver on 1000 modular-based SAT

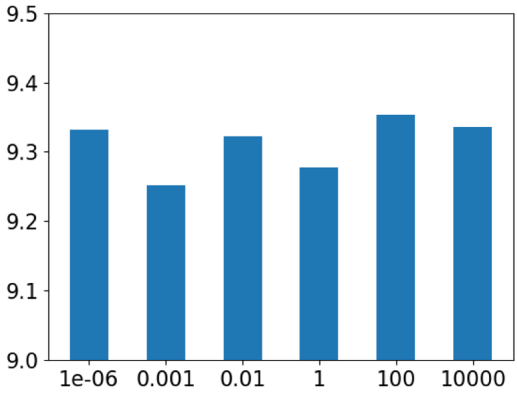


Fig. 2. Average runtime on an instance with CPPL using glucose on 200 modularity-based SAT instances for different values of ω with time in seconds on the y- and ω value on the x-axis.

instances. To have a meaningful comparison, the initial pool of parameterizations and the choice of parameterizations to run on the first instance is fixed to be the same for both approaches. The results given in Fig. 3 show that the CPPL ranking mechanism outperforms the ranking mechanism of ReACTR. We note that even though this result looks somewhat small, without generating new parameters we are dealing with only a randomly generated parameterization pool over the entire run of the algorithm.

4.5 RQ4: Direct Comparison of Performance of CPPL and ReACTR

We now compare the implementations of ReACTR with Glucose, CaDiCaL and CPLEX[1] For each experiment we use 16 Intel Xeon E5-2670 cores running at 2.6 GHz. In all results figures, we display a moving average of the runtimes of the parameterized solvers on the instances. Our results on three different benchmarks are shown in Fig. 4. The experiments with CaDiCaL and Glucose were executed three times and the runtime on each instance was averaged before plotting the comparison. The experiment with CPLEX could only be conducted once due to constrained time.

The computational time needed for ranking the parameterizations, choosing parameterizations that are to be run on the next instance and replacing parameterizations deemed to have bad performance is not included. Although we assume a realtime scenario, the practical applications we have in mind concede enough time between the arrival of instances for adjusting the pool and performing bookkeeping. Although CPPL is significantly more computationally

[1] A direct comparison of CPPL and ReACTR is not provided on the Glucose solver with the power-law SAT instance set. Even the first problem instance of this set could not be solved by Glucose within 24 h.

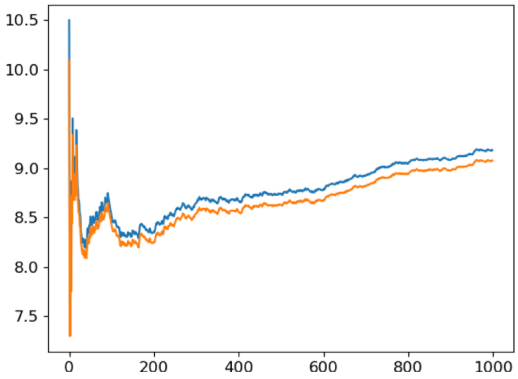
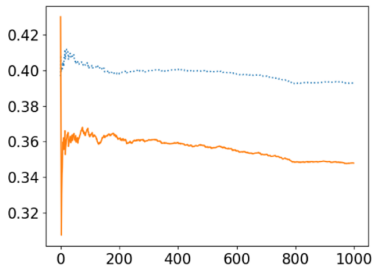


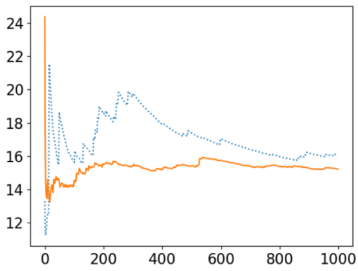
Fig. 3. Rolling average time of CPPL (orange) and ReACTR (blue) both without generating new parameterizations with time in seconds on the y- and number of instances on the x-axis. (Color figure online)

intensive than ReACTR, the time consumed by CPPL for the mentioned operations between instances, for example with CaDiCaL on the power-law SAT instance set, is on average only 0.4 s.

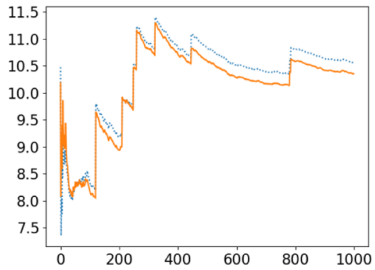
With CaDiCaL (Fig. 4a), CPPL shows considerably better performance than ReACTR on the modular-based SAT instance set. CaDiCaL shows advantageous properties for configuration purposes. The increase in complexity of the problem instances can be overcome by both approaches and the average solution time decreases with both. However, CPPL exploits CaDiCaL's properties further than ReACTR, which amounts in an advantage of approximately 10% of the total solution time compared with ReACTR. Regarding the comparison of the approaches in Fig. 4b, the results show that on the more difficult problem instance set, CPPL achieves better configuration of the solver. The solution time per instance of ReACTR varies in terms of the periodical changes in complexity of the instances. CPPL on the other hand seems to reliably find appropriate parameter configurations and the solution stays on a relatively stable level. The advantage regarding the total solution time of the instance set again amounts to approximately 10%. Figure 4c shows that the increase in difficulty of the instances influences the solution time for both approaches. However, CPPL solves the instances of the dataset about 160 s faster than ReACTR. Considering the average solution time of approximately 10 s per instance for ReACTR, this is an advantage of nearly two percent of the total solution time of the instance set. In Fig. 4d the CPPL approach with CPLEX does not outperform ReACTR. Despite an initial advantage on the instances the performance of the CPPL approach deteriorates. ReACTR outperforms CPPL by around one percent of the total runtime. We note that even if the experiments searching for good parameter values for CPPL showed only very small difference in performance, still choosing all values with best cumulative runtime resulted in a mostly good performance of CPPL.



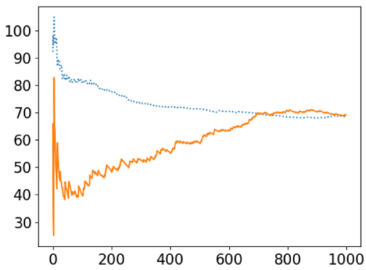
(a) Solver: CaDiCaL; Instances: SAT modular dataset.



(b) Solver: CaDiCaL; Instances: SAT power-law instance SAT set.



(c) Solver: Glucose; Instances: SAT modular dataset.



(d) Solver: CPLEX; Instances: MIP Frequency Assignment Problem.

Fig. 4. Comparison of CPPL (orange) and ReACTR (dashed blue) using several different solvers and instance sets with time in seconds on the y- and number of instances on the x-axis. (Color figure online)

5 Conclusion and Future Work

In this work, we considered the problem of realtime algorithm configuration and adapted the contextual preselection bandit method with the assumption of a Plackett-Luce ranking probability distribution to solve it, resulting in the CPPL realtime algorithm configurator. The approach allows for incorporating features of problem instances and parameterizations of algorithms and is competitive with the state of the art. Our first experimental results are promising.

In future work, we plan to further elaborate on different components of CPPL in order to fully exploit its potential. The feature engineering part, that is, the joint feature map $\Phi(\boldsymbol{x}, \boldsymbol{y})$ together with an embedding in a lower-dimensional space, appears to be especially critical in this regard. So far, we used a standard quadratic kernel for Φ and a simple PCA for dimensionality reduction, but there is certainly scope for improvement. Another direction concerns the idea of detecting and reacting to drift in a more active way. So far, an adaptation to changes in the data distribution is only accomplished implicitly through learning

in an incremental manner. In the case of abrupt changes, this adaptation is certainly not fast enough. Last but not least, we intend to have a deeper look at the different computational steps of the CPPL algorithm in order to reduce the algorithm's runtime.

Acknowledgements. This work was partly supported by the German Research Foundation (DFG) under grant HU 1284/13-1. This work was also partly supported by the German Federal Ministry of Economics and Technology (BMWi) under grant ZIM ZF4622601LF8. Moreover, the authors would like to thank the Paderborn Center for Parallel Computation (PC^2) for the use of the OCuLUS cluster.

References

1. Ansótegui, C., Malitsky, Y., Samulowitz, H., Sellmann, M., Tierney, K.: Model-based genetic algorithms for algorithm configuration. In: IJCAI (2015)
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), pp. 142–157 (2009)
3. Audemard, G.: Glucose and syrup in the SAT race 2015. In: SAT Competition 2015 (2015)
4. Balafrej, A., Bessiere, C., Paparrizou, A.: Multi-armed bandits for adaptive constraint propagation. In: IJCAI (2015)
5. Balcan, M.F., Sandholm, T., Vitercik, E.: Learning to Optimize Computational Resources: Frugal Training with Generalization Guarantees. arXiv preprint arXiv:1905.10819 (2019)
6. Bengs, V., Hüllermeier, E.: Preselection Bandits under the Plackett-Luce model. arXiv preprint arXiv:1907.06123 (2019)
7. Biedenkapp, A., Bozkurt, H.F., Eimer, T., Hutter, F., Lindauer, M.: Dynamic algorithm configuration: foundation of a new meta-algorithmic framework. In: ECAI (2020)
8. Biere, A.: CaDiCaL at the SAT race 2019. In: SAT Race 2019 - Solver and Benchmark Descriptions, p. 2 (2018)
9. Birattari, M., Stützle, T., Paquete, L., Varrentrapp, K.: A racing algorithm for configuring metaheuristics. In: Conference on Genetic and Evolutionary Computation (GECCO), pp. 11–18 (2002)
10. Birattari, M., Yuan, Z., Balaprakash, P., Stützle, T.: `F-Race` and iterated `F-Race`: an overview. In: Bartz-Beielstein, T., Chiarandini, M., Paquete, L., Preuss, M. (eds.) Experimental Methods for the Analysis of Optimization Algorithms, pp. 311–336. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-02538-9_13
11. Bischl, B., et al.: ASlib: a benchmark library for algorithm selection. Artif. Intell. **237**, 41–58 (2016)
12. Busa-Fekete, R., Hüllermeier, E., El Mesaoudi-Paul, A.: Preference-Based Online Learning with Dueling Bandits: A Survey. arXiv preprint arXiv:1807.11398 (2018)
13. Cheng, W., Hüllermeier, E., Dembczynski, K.: Label ranking methods based on the Plackett-Luce model. In: ICML, pp. 215–222 (2010)
14. Cicirello, V.A., Smith, S.F.: The max k-armed bandit: a new model of exploration applied to search heuristic selection. In: AAAI (2005)
15. El Mesaoudi-Paul, A., Bengs, V., Hüllermeier, E.: Online preselection with context information under the Plackett-Luce model. arXiv preprint arXiv:2002.04275 (2020)

16. Fitzgerald, T., Malitsky, Y., O'Sullivan, B.J.: ReACTR: realtime algorithm configuration through tournament rankings. In: IJCAI (2015)
17. Fitzgerald, T., Malitsky, Y., O'Sullivan, B.J., Tierney, K.: ReACT: real-time algorithm configuration through tournaments. In: Annual Symposium on Combinatorial Search (SoCS) (2014)
18. Friedrich, T., Krohmer, A., Rothenberger, R., Sutton, A.: Phase transitions for scale-free SAT formulas. In: AAAI, pp. 3893–3899 (2017)
19. Gagliolo, M., Schmidhuber, J.: Algorithm portfolio selection as a bandit problem with unbounded losses. Ann. Math. Artif. Intell. **61**, 49–86 (2011). https://doi.org/10.1007/s10472-011-9228-z
20. Giráldez-Cru, J., Levy, J.: A modularity-based random SAT instances generator. In: IJCAI, pp. 1952–1958 (2015)
21. Guo, S., Sanner, S., Graepel, T., Buntine, W.: Score-based Bayesian skill learning. In: Flach, P.A., De Bie, T., Cristianini, N. (eds.) ECML PKDD 2012. LNCS (LNAI), vol. 7523, pp. 106–121. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33460-3_12
22. Hoffman, M.D., Brochu, E., de Freitas, N.: Portfolio allocation for Bayesian optimization. In: UAI (2010)
23. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: LION, pp. 507–523 (2011)
24. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. J. Artif. Intell. Res. **36**, 267–306 (2009)
25. Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K.: Algorithm runtime prediction: methods & evaluation. Artif. Intell. **206**, 79–111 (2014)
26. IBM: CIBM ILOG CPLEX Optimization Studio: CPLEX User's Manual (2016). https://www.ibm.com/support/knowledgecenter/SSSA5P_12.7.0/ilog.odms.studio.help/pdf/usrcplex.pdf
27. Jamieson, K.G., Talwalkar, A.: Non-stochastic best arm identification and hyperparameter optimization. In: AISTATS, pp. 240–248 (2016)
28. Kleinberg, R., Leyton-Brown, K., Lucier, B.: Efficiency through procrastination: approximately optimal algorithm configuration with runtime guarantees. In: IJCAI, vol. 3, p. 1 (2017)
29. Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A.: Hyperband: a novel bandit-based approach to hyperparameter optimization. J. Mach. Learn. Res. **18**(1), 6765–6816 (2017)
30. Luce, R.D.: Individual Choice Behavior: A Theoretical Analysis. Wiley, Hoboken (1959)
31. Malitsky, Y., Sellmann, M.: Instance-specific algorithm configuration as a method for non-model-based portfolio generation. In: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR) (2012)
32. Maron, O., Moore, A.W.: Hoeffding races: accelerating model selection search for classification and function approximation. In: NIPS, pp. 59–66 (1993)
33. Phillips, M., Narayanan, V., Aine, S., Likhachev, M.: Efficient search with an ensemble of heuristics. In: IJCAI (2015)
34. Plackett, R.: The analysis of Permutations. J. Roy. Stat. Soc. Ser. C (Appl. Stat.) **24**(1), 193–202 (1975)
35. Santos, H., Toffolo, T.: Python MIP: Modeling Examples (2018–2019). https://engineering.purdue.edu/~mark/puthesis/faq/cite-url/. Accessed 23 Jan 2020

36. Schäfer, D., Hüllermeier, E.: Dyad ranking using Plackett-Luce models based on joint feature representations. Mach. Learn. **107**(5), 903–941 (2018). https://doi.org/10.1007/s10994-017-5694-9
37. Shang, X., Kaufmann, E., Valko, M.: A simple dynamic bandit algorithm for hyperparameter tuning. In: Workshop on Automated Machine Learning at ICML, June 2019
38. St-Pierre, D.L., Teytaud, O.: The Nash and the bandit approaches for adversarial portfolios. In: IEEE Conference on Computational Intelligence and Games, pp. 1–7 (2014)
39. Tavakol, M., Mair, S., Morik, K.: HyperUCB: hyperparameter optimization using contextual bandits (2019)
40. Wang, J., Tropper, C.: Optimizing time warp simulation with reinforcement learning techniques. In: Winter Simulation Conference, pp. 577–584 (2007)
41. Yue, Y., Joachims, T.: Interactively optimizing information retrieval systems as a dueling bandits problem. In: Proceedings of International Conference on Machine Learning (ICML), pp. 1201–1208 (2009)