Kemminer, Robin; Lange, Jannick; Kempkes, Jens Peter; Tierney, Kevin; Weiß, Dimitri

Article — Published Version

Configuring Mixed-Integer Programming Solvers for Large-Scale Instances

Operations Research Forum

Suggested Citation: Kemminer, Robin; Lange, Jannick; Kempkes, Jens Peter; Tierney, Kevin; Weiß, Dimitri (2024) : Configuring Mixed-Integer Programming Solvers for Large-Scale Instances, Operations Research Forum, ISSN 2662-2556, Springer International Publishing, Cham, Vol. 5, Iss. 2, https://doi.org/10.1007/s43069-024-00327-7

Operations Research Forum (2024) 5:48
https://doi.org/10.1007/s43069-024-00327-7

RESEARCH

Configuring Mixed-Integer Programming Solvers for Large-Scale Instances

Robin Kemminer[1] · Jannick Lange[1] · Jens Peter Kempkes[1] · Kevin Tierney[2] · Dimitri Weiß[2]

Received: 6 April 2023 / Accepted: 30 April 2024 / Published online: 30 May 2024

Abstract
Algorithm configuration techniques automatically search for parameters of solvers and algorithms that provide minimal runtime or maximal solution quality on specified instance sets. Mixed-integer programming (MIP) solvers pose a particular challenge for algorithm configurators due to the difficulty of finding optimal, or even feasible, solutions on the large-scale problems commonly found in practice. We introduce the OPTANO Algorithm Tuner (OAT) to find configurations for MIP solvers and other optimization algorithms. We present and evaluate several critical components of OAT for solving MIPs in particular and show that OAT can find configurations that significantly improve the solution time of MIPs on two different datasets.

Keywords Algorithm configuration · Mixed-integer programming · Large-scale problem instances

✉ Kevin Tierney
kevin.tierney@uni-bielefeld.de

Robin Kemminer
robin.kemminer@optano.com

Jannick Lange
jannick.lange@optano.com

Jens Peter Kempkes
jens.peter.kempkes@optano.com

Dimitri Weiß
dimitri.weiss@uni-bielefeld.de

1 OPTANO GmbH, Technologiepark 18, Paderborn 33100, NRW, Germany

2 Decision and Operation Technologies, Bielefeld University, Universitätsstraße 25, Bielefeld 33615, NRW, Germany

1 Introduction

The performance of algorithms and solvers varies greatly depending on the settings of the *parameters* controlling the behavior of the approach [1]. In particular, parameter settings that work well for a particular dataset of instances may work poorly on a different dataset, especially in terms of special problem structures or instance sizes. To ensure good performance of an algorithm, either in terms of runtime or solution quality, it is critical that algorithm parameters be *configured* or *tuned* for the types of instances the algorithm is expected to solve in practice.

Searching for high-quality parameter settings by hand is a time-consuming endeavor, hence several tools have been developed to *automatically* determine good parameter settings for a solver or algorithm given a dataset of instances. These tools use a variety of methods ranging from fractional factorial design [2], local search [3, 4], genetic algorithms [5–7], Bayesian optimization [8, 9], and racing [10] (see [1] for a full overview). Most *algorithm configurators* support configuring for one or both of the following settings: (1) minimization of *target algorithm* runtime or (2) maximization of *solution quality*. Some target algorithms, such as mixed-integer programming solvers, require a mixture of configuring for runtime and solution quality to find high quality solutions [6] to effectively tune their parameters for a given dataset.

Mixed-integer programming (MIP) solvers can tackle a wide range of problem types and thus ought to be configured for the instance set they are meant to solve. Indeed, with this in mind, IBM CPLEX, Gurobi, and FICO Xpress, three of the most well-known general MIP solvers, have built in parameter tuning capabilities [11–13]. Moreover, MIP solvers have been a focus of the algorithm configuration (AC) community for some time, with early results providing speed-ups of up to 52x on the CPLEX solver [14] and recent results showing there are still performance gains to be had in tuning these approaches [15, 16].

Most of the successes of AC solvers on MIP have involved small-scale problems; however, in industry, problems with tens of thousands or even millions of variables must be solved on a regular basis. These extremely large problems post a challenge to configurators. On the one hand, when tuning for runtime, many instances will likely not finish in the given timeout, leading to wasted executions and poor performance of the configurator. On the other hand, when tuning for solution quality, many MIP runs may not find feasible solutions, meaning a mechanism for comparing these failed executions is required to provide the configurator with a search trajectory.

This paper introduces the OPTANO Algorithm Tuner (OAT), a general algorithm configurator that has a special focus on addressing large-scale MIP instances. The contributions are as follows:

- We describe OAT, an AC tool based on the GGA algorithm.
- We investigate a *dominance racing* mechanism to shorten configuration times on MIP without sacrificing overall performance.
- We further show on a real world problem that configuring smaller copies of large instances (i.e., instances of reduced size, but similar structures to large instances) is effective for finding good configurations for the large instances.

We make OAT freely available under the MIT license at https://github.com/OPTANO/optano.algorithm.tuner.

This paper is organized as follows: We discuss the current state-of-the-art for configuring MIP solvers in Section 2. In Section 3, we describe OAT, which forms the experimental basis for this work followed by the extensions of OAT specifically for configuring MIP solvers. We evaluate the extensions computationally in Section 4 and show that OAT can find high-quality configurations for a large-scale, real-world MIP dataset. Finally, we discuss future work and conclude in Section 5.

2 Related Work and Background Information

We provide a general overview of AC, including offline and realtime AC, and discuss its application to configure MIP solvers. For further details about AC and related problem settings, we refer interested readers to [1].

2.1 Offline Automated AC

We first formalize the offline AC problem and adopt the notation in [3]. The goal of AC is to optimize the performance of a parameterized algorithm $\mathbb{A}$. To achieve this, the configurator searches for high-quality parameter configurations θ in the space of all possible configurations Θ of $\mathbb{A}$. The quality of a configuration is measured by a performance metric m on a set of problem instances $\Pi \subseteq \hat{\Pi}$, where $\hat{\Pi}$ represents the full distribution of problem instances and Π the sample the AC approach is provided, such that $m : \hat{\Pi} \times \Theta \to \mathbb{R}$. The general process of algorithm configuration is depicted in Fig. 1.

Offline AC aims at finding a high quality configuration θ^* that performs well over any possible set Π drawn from $\hat{\Pi}$. To this end, a set of problem instances Π, called the training set, is drawn from $\hat{\Pi}$ is provided to the AC method that is representative of $\hat{\Pi}$. The configuration space Θ is searched for high quality configurations θ_i on the training set, where the aim is to minimize $\sum_{\pi \in \hat{\Pi}} m(\pi, \theta)$ in the runtime scenario, whereas in the solution quality scenario, this term is maximized.

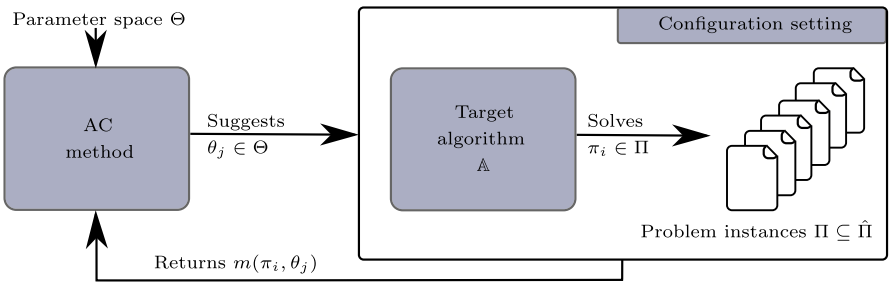


Fig. 1 The information flow of offline automated AC

Several well-known approaches have been developed for offline AC using both model-based (i.e., machine learned models to predict/evaluate configurations) and non-model-based approaches. ParamILS [3], a non-model-based approach, employs an iterated local search combined with an adaptive capping mechanism to avoid wasting CPU time on poorly performing configurations. The AC method on which our approach in this paper is based, GGA [5, 7], uses a genetic algorithm with a tournament-based racing mechanism, while irace [10] also uses racing, but in a statistical fashion. GPS [4] exploits parameter configuration landscape structures and examine sparameters in a semi-independent way. Model-based configurators include SMAC [8], which is based on a Bayesian optimization paradigm that uses a random forest to predict the performance metric of a given configuration, and GGA++ [6], which uses a random forest with a modified tree building mechanism to predict configuration performance.

2.2 Algorithm Configuration for MIP

The AC community has long targeted the MIP setting due to the long runtimes of solving MIP instances and the industrial relevance of MIPs. Several commercial MIP solvers include configuration procedures, such as CPLEX [11], Gurobi [12], and FICO Xpress [13], although we note that these have not been shown to be more effective than any AC method in the literature.

ParamILS is used to configure the solvers CPLEX, Gurobi, and LpSolve in [14], resulting in significant speedups on seven different instance sets. Several MIP settings are included in the AClib [17], allowing developers of AC methods to test on standard benchmarks. AC methods have also been used to configure MIPs in instance-specific settings, i.e., a specific configuration θ is assigned to each instance in $\hat{P}i$, e.g., in [18] using the Hydra method [19] and in [20] using the instance-specific algorithm configuration (ISAC) approach. We further note that online/dynamic approaches for configuring MIP parameters exist, e.g., DASH [21] (see also dynamic AC (DAC), [22]).

3 OPTANO Algorithm Tuner

OAT is a general algorithm configurator distributed as a.NET nuget package that can be used as a standalone configurator or integrated directly into solvers or algorithms written in.NET. The goal of OAT is to provide a configurator that has state-of-the-art performance combined with the reliability expected of software running in production. While OAT is originally based on GGA [5] and GGA++ [6], it has since been extended to include search strategies based on JADE [23] and active CMA-ES [24]. OAT is inherently distributed and can run its target algorithm in parallel across multiple machines to reduce the overall wall-clock time of the configuration process. In addition, OAT supports configuring in multiple sessions, allowing the configuration process to be restarted should it be interrupted by, e.g., a system failure or reaching a resource limit. Finally, OAT includes numerous ideas from the literature, including parameter tree customization (from GGA), non-numeric evaluation metrics (GGA++), and adaptive capping strategies (ParamILS). We first describe how OAT works and

describe how it distributes jobs across cores, which differs from previous distributed versions of GGA. Then, we introduce its MIP-specific enhancements, namely the new evaluation metric and short-circuit domination rule.

3.1 OAT's Configuration Process

OAT is based on the genetic algorithm-based GGA and GGA++ configurators from a methodological standpoint, however not an engineering one. The function of OAT consists of three phases: (1) initialization, (2) the main loop, and (3) convergence/termination. The main loop iterates until OAT either reaches the maximum number of generations (as specified by the user), a maximum number of evaluations of the target algorithm, or runs out of time. Figure 2 provides an overview of the function of OAT, and we refer readers to [5] and [6] for further details.

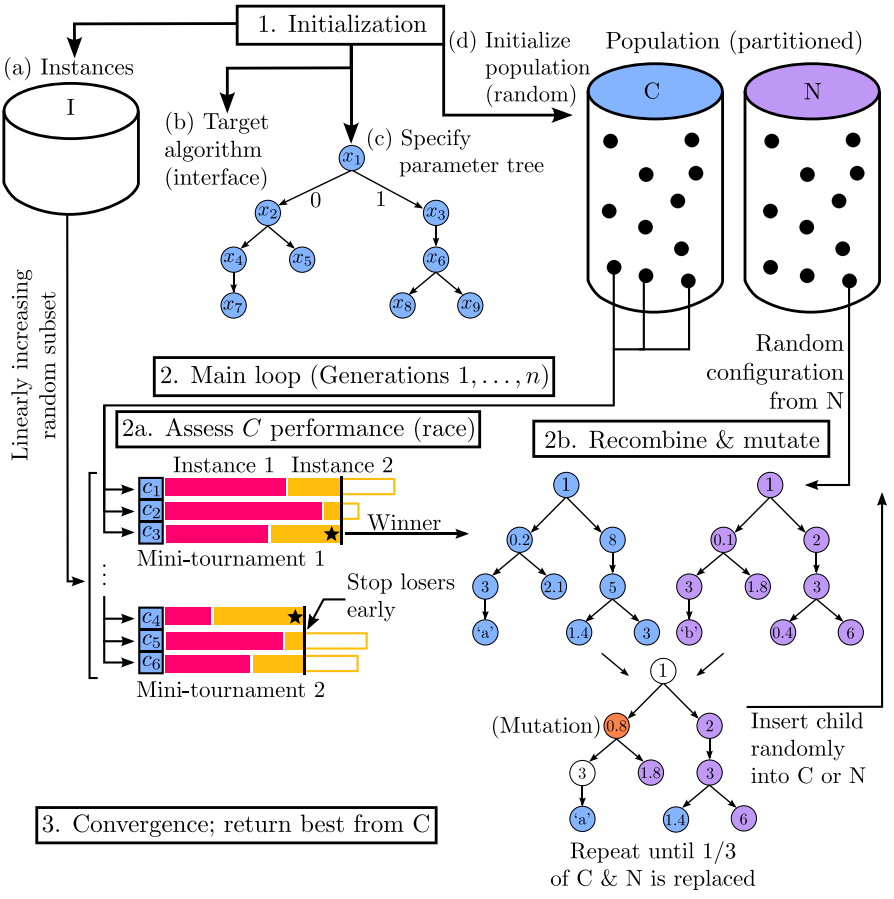


Fig. 2 Overview of the GGA approach [5] used in OAT

Initialization Considering the previously introduced formalization of algorithm configuration, OAT needs the following four inputs to start its search. First, it needs a list of instances, I, that will be investigated, potentially associated with random seeds. Second, OAT must be told how to invoke the target algorithm, either on the command line or through an interface into other.NET code. Third, the target algorithm parameters to be configured must be specified. OAT takes a structured view of parameters as in GGA, accepting a *parameter tree* defining relations between parameters (see part 1(c) of Fig. 2). For example, Gurobi [12] contains several parameters relating to the presolver that can be adjusted to change its behavior. Another parameter turns the presolver on and off, meaning that the parameters relating to the presolver *depend* on the parameter to turn it on and off. This information is used during search to generate new configurations. Thus, the dependent parameters are placed lower below the presolver on/off parameter, and the recombination procedure takes this into account when creating new individuals. Finally, OAT's own internal parameters can be changed from their default values, relating to how its search strategy functions.[1]

Given the inputs outlined above, OAT initializes a population consisting of the default configuration(s) and randomly generated configurations, partitioned into two groups, representing the *competitive* (C) configurations that will be run on the target algorithm, and *non-competitive* (N) configurations, which act as a diversity store.

Main Loop This phase consists of up to *n generations*, in which configurations from the C population are assessed in races and the winners are recombined with non-competitive configurations. At the beginning of each generation, a subset of instances are sampled from the instance pool. This subset linearly increases with each generation until either all instances are used or a user-specified maximum value is reached. All configurations of the competitive population must be evaluated on the currently active subset of instances. We note that some configurations may have already been evaluated on some of the instances in previous generations; these configurations need not be evaluated on the same instances again. If the size of the competitive population is larger than the number of available CPUs,[2] the configurations are split into *mini-tournaments* equal to the number of available CPUs. In the pure runtime setting, mini-tournaments are executed until a fixed percentage of the configurations have solved all instances; in the case of Fig. 2, only one configuration can win the race. The rest of the configurations are subsequently terminated when they have used the same amount of CPU time as the winning configuration(s). The mechanism by which OAT distributes mini-tournaments is described in more detail below. In the case of MIP, we slightly modify this procedure and describe this in Sections 3.2 and 3.3.

After all planned evaluations for the current generation are completed, the population is updated based on the performance of the configurations. Several options are available to do this, such as the GGA crossover mechanism in GGA, the genetic

[1] We note that "tuning the tuner" poses a significant challenge; thus, these parameters are set to values that have worked well for GGA and GGA++ in the past.

[2] We assume the target algorithm is single threaded in our descriptions; however, OAT also supports configuring multithreaded target algorithms.

engineering algorithm of GGA++, as well as approaches based on JADE and active CMA-ES. Figure 2 shows the GGA crossover mechanism in which the winners of the mini-tournaments are recombined with randomly chosen members of the non-competitive population. The crossover procedure constructs a new configuration by randomly choosing components from the two parents. We refer to [5] for the full details of this algorithm and of the subsequent mutation operator. A specified percentage of the population is replaced every generation (usually one third) through the recombination procedure in the hope of generating high-quality configurations. Model-based recombination is also possible in OAT using the GGA++ recombination strategy, see [6] for details.

Termination The main loop of OAT runs until one of three conditions is reached. The first condition is whether the maximum number of generations is achieved (usually 75 or 100). The second condition is whether a maximum number of evaluations of the target algorithm is exceeded. Finally, the third condition is whether the maximum wall-clock time of the configurator is exceeded. Note that GGA supports a convergence criterion that checks whether the population is improving or not, but this is not implemented yet in OAT.

Increasing Mini-Tournament CPU Utilization The mini-tournaments as described above must be efficiently distributed across the available CPU resources. A key engineering advancement of OAT over previous GGA configurators is that it attempts to maximally utilize available CPU resources. While OAT runs mini-tournaments to race competitive configurations, it distributes mini-tournaments across multiple nodes according to a priority queue of configuration-instance-seed tuples that must still be run, leading to less wasted CPU capacity than, e.g., GGA and GGA++. OAT prioritizes configurations that it believes are likely to finish first so that the finishing time can be used in the short-circuit evaluation of other configurations according to the formula

$$priority(c) = 100\left(\frac{timeouts(c)}{|I_g|}\right) + 10\left(\frac{running(c)}{|I_g|}\right) + \frac{runtime(c)}{\kappa|I_g|},$$

where c is a configuration being evaluated in the current generation, g, $timeouts(c)$ provides the number of timeouts the configuration c has had in the current generation so far, I_g is the instance subset being considered in the current generation, $running(c)$ describes the number of instance-seed pairs c is currently running on, $runtime(c)$ gives the total runtime of c so far in the current generation, and κ is the timeout as previously defined.

The proposed mechanism runs configuration-instance-seed tuples with a low value. The intuition is that a low priority score corresponds first to configurations with a low number of timeouts, following that configurations that have not yet seen much CPU time are favored, and finally, the total runtime expended should be a low percentage of the total CPU time allotment for the configuration. In this way, configurations are preferred that are likely to finish the mini-tournaments first, allowing us to dominate poor-performers before they waste CPU resources (see Section 3.3).

3.2 MIP Evaluation Metric

The evaluation metric tells OAT how to interpret and aggregate the performance of the target algorithm on a subset of the training instances. One of the main considerations when developing a runtime evaluation metric is how to deal with timeouts. While many configurators simply use the average performance of a configuration over a set of instances, this does not significantly discourage timeouts from occurring. Hence, many configurators also support the so-called *PAR10* score, which extends the average by multiplying timeouts by a factor of 10.

While PAR10 effectively penalizes timeouts, when an instance set contains many difficult instances, it often does not offer effective search guidance. To improve on this, in the gray-box configuration schemes [25] and [16], we analyze intermediate output of the target algorithm to assist in ranking or otherwise scoring timeouts. In the case of CPPL, ties between configurations that do not finish are broken using the quality of the feasible solution found (if one was found).

In contrast to realtime configuration, where only a single instance is solved per iteration, in offline configuration, breaking times is somewhat more complicated. Especially in the first few iterations of configuration, timeouts are very likely as the search process has not yet identified good configurations. Hence, it is critical to have an effective mechanism for comparing configurations even if none find optimal solutions to the instances being solved. Thus, to minimize the runtime of solving MIPs, we propose the following simple ranking scheme. Assume we are given two configurations A and B that are run on n instances and the following rules are applied in order:

1. If A finds more feasible solutions than B, A is better.
2. Otherwise, if A has less timeouts than B, A is better.
3. Otherwise, if A has a lower average MIP gap over the timeout runs than B, A is better.
4. Otherwise, if A has a lower average runtime than B, A is better.

Since the runtime is a floating point value, and there is generally some noise in its measurement, this ranking is all but guaranteed to return a total order over the available configurations. Note that the focus of the ranking is on feasibility and not optimality. The reason for this is that companies solving MIPs in practice would much rather have feasible solutions for all of the instances they are investigating than optimal solutions on a few and no solution at all on the rest. However, while our motivation for these rule set is a practical one, we show later that there is also a computational benefit to the rules, as these rules help guide the configurator's search towards areas of the search space with configurations effective at finding optimal solutions.

3.3 Dominance Racing

Running MIPs is computationally expensive; thus, if we detect that a particular configuration is dominated, we can stop running it and use the available resources to run something else. GGA and GGA++ accomplish this in the average or PAR10 runtime setting through their racing mechanism, which ensures that configurations that are dominated are stopped before wasting CPU resources. However, when using a rank-

ing mechanism for MIP, we need to adjust the domination criteria to avoid wasting CPU time.

The goal of our short circuit evaluation is to ensure that configurations with no chance of winning their mini-tournament are stopped as soon as this is detected. Given a mini-tournament, once one of the configurations finishes an instance, we can then check if any other configurations in the mini-tournament are dominated. Let the current best configuration of the tournament be A, and without loss of generality, another configuration in the mini-tournament that is not yet finished be B. Let all unfinished instances[3] of A be considered timeouts for the purpose of the domination, and let all unfinished instances of B be represented as optimal solutions found immediately. Then, using our ranking mechanism, rank A and B. If B is ranked worse than A, we know that B will never be better than A and can be eliminated from consideration.

4 Experimental Results

We evaluate OAT on a set of synthetic MIP instances that model the frequency assignment problem from [26] followed by a real-world instance set modeling a strategic network planning problem for a customer of OPTANO GmbH. We address the following two research questions:

- RQ1: Does the dominance racing allow OAT to find the same or better configurations in less wall-clock time than without dominance racing?
- RQ2: Can OAT find high-quality configurations for small datasets of long-running MIPs?

In the following, we configure OAT using the GGA++ search strategy on the target algorithm Gurobi 8.1 on two Intel Xeon E5-2680 processors with 12 cores each running at 2.5 GHz and 256 GB of RAM.

4.1 RQ1: Effectiveness of Dominance Racing

We examine the effectiveness of the dominance racing mechanism using the proposed ranking method and with the standard PAR10 metric on a dataset of synthetic instances representing the frequency assignment problem [26]. We configure OAT for 100 generations and increase the number of instances in each generation linearly until generation 75, after which all instances are run in each generation. We allow Gurobi to use a single thread. The dataset of instances is split into 25 training instances assigned to 2 seeds each, and a test set of 25 instances with 10 seeds each to try to avoid erraticism/variability in solving the instances [27] from influencing the results. The timeout for Gurobi is set to 300 s. We repeat this experiment three times.

Table 1 shows the average results over three runs of OAT to tune the synthetic frequency assignment problem instances, using PAR10 with and without the dominance racing mechanism, and the ranking metric with it. RR stands for runtime racing,

[3] With *unfinished instances*, we refer to any instance that the configuration has not yet been run on or is not finished running on.

Table 1 Average target algorithm evaluations and runtime of OAT, SMAC and grbtune over three executions of OAT and 72 executions of SMAC and grbtune (leading to an equivalent total CPU-time allotment[a]), along with the resulting performance of the configurations on Gurobi on average over these executions. The evaluation time of Gurobi with its default parameters on the instance set is also provided

	Config. (OAT)		Gurobi (Target Algorithm) After 72h [s]		After 100g [s]	
	Evals.	Wall [h]	Train	Test	Train	Test
Default	-	-	288.7	308.8	288.7	308.8
grbtune	-	-	377.7	395.7	-	-
SMAC (PAR10)	-	-	155.9	145.1	-	-
OAT (PAR10 + RR)	82031	187.5	93.4	82.8	73.9	71.4
OAT (PAR10 + DR)	48656	92.9	72.5	71.4	72.5	71.4
OAT (Ranking + DR)	43788	84.0	74.3	76.1	70.5	74.5

[a]We use the single-core version of SMAC and grbtune; hence, we run it 72 times to have an equivalent CPU-time allotment to our three executions of OAT on 24 cores

which is the type of racing used by GGA++ in which a tournament is stopped once enough configurations finish. By definition, this is a special case of dominance racing (DR). While dominance racing is more aggressive than runtime racing, it also does not eliminate any run of a potential tournament winner. Since the results of PAR10 give a strong impression and these experiments are computationally intensive, we refrain from combining ranking with runtime racing, but focus on ranking with dominance racing. Dominance racing is able to cut the overall wall-clock time to configure with OAT by nearly half, without sacrificing any performance on the test set using both PAR10 and the ranking metric after 100 generations. If OAT is stopped after 72 h, the performance of DR is superior to not using it on both the training and test sets. It is thus clear that many of the target algorithm executions made are actually avoidable. Combining the ranking metric with DR leads to even faster configuration than using PAR10; however, the test performance both after 72 h and 100 generations shows signs of overfitting. We note that DR itself cannot result in overfitting, so this is potentially due to the ranking mechanism being overly aggressive. Finally, we note that regardless of the configuration of OAT, large gains over the default parameters are visible.

OAT performs well compared to SMAC and grbtune thanks to the addition of RR and DR, which we are unable to add to these configurators. We note that grbtune, the built in parameter tuner of Gurobi, actually finds configurations that perform worse than the default configuration on average. In about half the runs of grbtune, the default configuration was returned. However, in the other half, poorly performing configurations were returned that did not generalize to the test set. As the implementation of grbtune is private, we are unable to suggest why this is the case.

The three configurations found by OAT agree with default parameters of Gurobi roughly one third of the time. Note, however, that which parameters are the same as the defaults vary between the configurations greatly. The agreement between the three configurations ranges from 26% of the parameters the same up to 40%. Note that there is only one floating point parameter, and it is different for all configurations. Furthermore, the parameters with large integer ranges tend to be significantly different

Table 2 Runtime until the first solution is found in minutes on the large instance test set

	Default	All instances	LOO
Average	42.7	1.9	4.9
Median	11.5	2.0	6.0

from each other. The key insight from this analysis is that even configurations with significantly different configurations can have similar performance. While Gurobi includes many parameters that can be set to "automatic" values (i.e., Gurobi uses a heuristic to decide its value in an instance-specific way), our configurations only use this option between 21 and 28% of the time.

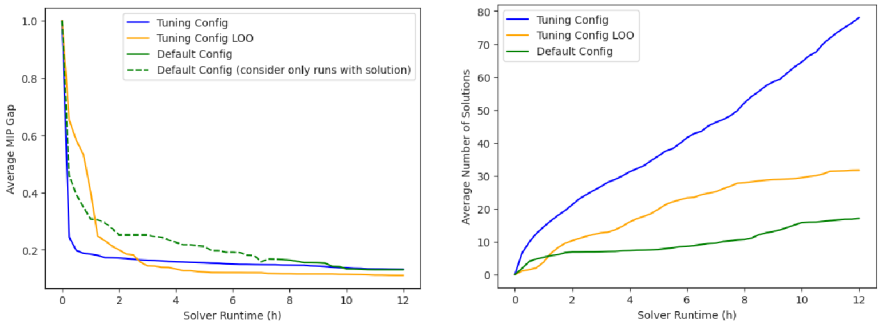
4.2 RQ2: Configuring Large-Scale MIPs

We now configure a small dataset of large-scale, real MIP instances from a project at OPTANO GmbH. The instances represent a strategic network planning problem for production and delivery for a customer who cannot be named. The instances contain over 470,000 variables, over 50,000 of which are integer. Furthermore, the instances have over 2.5 million non-zeros, making them practically impossible to solve to optimality even given days of computation time. Due to the strategic nature of the problem, there are only three instances available, and they are extremely hard to solve. The goal is to find a good configuration to solve these three instances as well as future instances that the customer may need solved.

To configure this dataset, we first create three *small-scale copies* of the three instances. While we are unable to propose a general method for doing this, on many problems, it is possible to create smaller copies that maintain the original structure of the instance. For the network planning problem at hand, we generate smaller instances by restricting the degrees of freedom in the optimization model. For example, we decide a priori which demands will be served and which machines will be used for those demands. Thus, we will configure the small instances and apply the resulting configurations on the larger instances.

Due to the small size of the dataset, we attempt to mitigate the effect of randomness on our results so that we can draw some limited conclusions about tuning small-scale copies of MIPs. To this end, we use leave-one-out (LOO) cross validation and perform four different configurations of the small instances, one including all three instances, and three more with all combinations of two out of three instances. Validation is performed on the large instances; in this way, we simulate the situation where a new large instance is encountered after configuration is completed. We assign four random seeds to each training instance and assign 10 to each large instance in the test set. We again configure Gurobi 8.1, but this time allow Gurobi to use two threads as this is how the customer's environment is set up. We configure for 50 generations and use the entire set of instance seed pairs in all generations. The timeout is set to 30 s.

We configure and evaluate the four settings as discussed and aggregate the results in Table 2, which shows the average time in minutes until the first solution is found on



(a) Average MIP gap. (b) Average number of solutions.

Fig. 3 Visualizations over the course of solving the instances of the test set

the test set. We note that the default parameters were not particularly robust, with two out of thirty instance-seed pairs requiring almost 8 h to find a first solution, which is unacceptable for implementing the model for the customer. The configurations found by OAT are significantly more robust, with the maximum time needed to find a first solution only 4.0 min (all instances) resp. 9.0 min (LOO).

Figure 3a and b provide information over the course of the execution of the instances of the test set. In Fig. 3a, the configuration tuned on all three instances lowers the MIP gap the fastest, but then levels out. Surprisingly, the LOO configurations catch up before the three hour mark and lower the MIP gap further, although we note this is very likely just noise. Nonetheless, the fact that configuring on two out of three instances has similarly to configuring on all three instances is promising information for the many real-world domains where often very few instances are available. Furthermore, Fig. 3b shows that the all configurations found generate more solutions than the default Gurobi configuration. While the number of solutions alone is not an indication of quality (they could all be bad), in general, when solving MIPs, finding solutions is good for users who can receive intermediate feedback at multiple intervals.

5 Conclusion and Future Work

We introduced OAT, a general-purpose algorithm configuration tool with special mechanisms for configuring MIP solvers. We show that using the proposed ranking mechanism and short-circuit domination rule significantly reduces the time required to configure MIPs on a synthetic dataset. Furthermore, we confirm that OAT is capable of configuring a real-world strategic network design problem that has since been deployed at a customer of OPTANO. There are still many open research questions for future work, such as how to automatically reduce the size of large MIP instances so that they can be effectively tuned or how to further avoid wasting time on unpromising configurations by, e.g., applying machine learning models.

Acknowledgements The authors thank Sebastian Milz for his support of our industrial test case. The authors would also like to thank the Paderborn Center for Parallel Computation (PC^2) for the use of the Noctua cluster.

Author Contributions RK and JL coded OAT under the supervision of JPK and KT. DW supported the writing of the paper, which was mainly carried out by KT and RK. All authors read and approved the final manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL. The authors are supported in part by the funding program *Zentrales Innovationsprogramm Mittelstand (ZIM)* (Grant No. ZF4622601LF8) of the German Federal Ministry for Economic Affairs and Climate Action, and the project *Maschinelle Intelligenz für die Optimierung von Wertschöpfungsnetzwerken (MOVE)* (Grant No. 005-2001-0042) of the "it's OWL" funding of the Ministry of Economics, Innovation, Digitalization and Energy of the German state of North Rhine-Westphalia.

Code Availability The source code of the OPTANO Algorithm Tuner is available at https://github.com/OPTANO/optano.algorithm.tuner.

Declarations

Ethics Approval Not applicable.

Consent to Participate Not applicable.

Consent for Publication Not applicable.

Competing Interest The authors declare no competing interests.

References

1. Schede E, Brandt J, Tornede A, Wever M, Bengs V, Hüllermeier E, Tierney K (2022) A survey of methods for automated algorithm configuration. J Artif Intell Res 75:425–487
2. Adenso-Díaz B, Laguna M (2006) Fine-tuning of algorithms using fractional experimental designs and local search. Oper Res 54:99–114
3. Hutter F, Hoos HH, Leyton-Brown K, Stützle T (2009) Paramils: an automatic algorithm configuration framework. J Artif Intell Res(JAIR), p 267–306
4. Pushak Y, Hoos H (2020) Golden parameter search: exploiting structure to quickly configure parameters in parallel. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), p 245–253. https://doi.org/10.1145/3377930.3390211
5. Ansótegui C, Sellmann M, Tierney K (2009) A gender-based genetic algorithm for the automatic configuration of algorithms. In: Principles and Practice of Constraint Programming - CP 2009, p 142–157. https://doi.org/10.1007/978-3-642-04244-7_14
6. Ansótegui C, Malitsky Y, Samulowitz H, Sellmann M, Tierney K (2015) Model-based genetic algorithms for algorithm configuration. In: International Joint Conferences on Artificial Intelligence Organization (IJCAI)

7. Ansótegui C, Pon Farreny J, Sellmann M, Tierney K (2021) PyDGGA: distributed GGA for automatic configuration, p 11–20. https://doi.org/10.1007/978-3-030-80223-3_2
8. Hutter F, Hoos HH, Leyton-Brown K (2011) Sequential model-based optimization for general algorithm configuration. In: Learning and Intelligent Optimization (LION), p 507–523
9. Lindauer MT, Eggensperger K, Feurer M, Biedenkapp A, Deng D, Benjamins C, Sass R, Hutter F (2022) SMAC3: a versatile Bayesian optimization package for hyperparameter optimization. J Mach Learn Res 23:54–1549
10. López-Ibáñez M, Dubois-Lacoste J, Stützle T, Birattari M (2016) The irace package: iterated racing for automatic algorithm configuration. Oper Res Perspect, p 43–58. https://doi.org/10.1016/j.orp.2016.09.002
11. IBM. IBM CPLEX User's manual for CPLEX. https://www.ibm.com/docs/en/icos/22.1.1?topic=optimizers-users-manual-cplex. Accessed 27 Apr 2023
12. Gurobi. Gurobi optimization documentation. https://www.gurobi.com/documentation/. Accessed: 27 Apr 2023
13. FICO. Fico FICO Xpress optimization help. https://www.fico.com/fico-xpress-optimization/docs/latest/solver/optimizer/HTML/GUID-3BEAAE64-B07F-302C-B880-A11C2C4AF4F6.html. Accessed 27 Apr 2023
14. Hutter F, Hoos HH, Leyton-Brown K (2010) Automated configuration of mixed integer programming solvers. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR), p 186–202. Springer
15. Mesaoudi-Paul E, Weiß D, Bengs V, Hüllermeier E, Tierney K et al (2020) Pool-based realtime algorithm configuration: a preselection bandit approach. In: International Conference on Learning and Intelligent Optimization, p 216–232. Springer
16. Weiß D, Tierney K (2022) Gray box realtime algorithm configuration. In: Learning and Intelligent Optimization. Springer
17. Hutter F, López-Ibánez M, Fawcett C, Lindauer M, Hoos HH, Leyton-Brown K, Stützle T (2014) AClib: a benchmark library for algorithm configuration. In: Learning and Intelligent Optimization: 8th International Conference, LION 8, p 36–40. Springer
18. Xu L, Hutter F, Hoos HH, Leyton-Brown K (2011) Hydra-MIP: automated algorithm configuration and selection for mixed integer programming. In: RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Artificial Intelligence (IJCAI), p 16–30
19. Xu L, Hoos H, Leyton-Brown K (2010) Hydra: automatically configuring algorithms for portfolio-based selection. In: Proceedings of the AAAI Conference on Artificial Intelligence 24:210–216
20. Kadioglu S, Malitsky Y, Sellmann M, Tierney K (2010) ISAC - instance-specific algorithm configuration 215:751–756. https://doi.org/10.3233/978-1-60750-606-5-751
21. Liberto GD, Kadioglu S, Leo K, Malitsky Y (2016) Dash: dynamic approach for switching heuristics. Eur J Oper Res 248(3):943–953. https://doi.org/10.1016/j.ejor.2015.08.018
22. Biedenkapp A, Bozkurt HF, Eimer T, Hutter F, Lindauer M (2020) Dynamic algorithm configuration: foundation of a new meta-algorithmic framework. In: ECAI 2020, p 427–434. IOS Press
23. Zhang J, Sanderson AC (2009) Jade: adaptive differential evolution with optional external archive. IEEE Trans Evol Comput 13(5):945–958
24. Jastrebski GA, Arnold DV (2006) Improving evolution strategies through active covariance matrix adaptation. In: 2006 IEEE International Conference on Evolutionary Computation, p 2814–2821. IEEE
25. El Mesaoudi-Paul A, Weiß D, Bengs V, Hüllermeier E, Tierney K (2020) Pool-based realtime algorithm configuration: a preselection bandit approach. Lecture Notes in Computer Science, vol. 12096, p 216–232. Springer. https://doi.org/10.1007/978-3-030-53552-0_22
26. Anderson LG (1973) A simulation study of some dynamic channel assignment algorithms in a high capacity mobile telecommunications system. IEEE Trans Veh Technol 22(4):210–217. https://doi.org/10.1109/T-VT.1973.23553
27. Fischetti M, Monaci M (2014) Exploiting erraticism in search. Oper Res 62(1):114–122