Exploiting Configurations of MaxSAT Solvers

Josep Alòs ✉ 🆔
Logic & Optimization Group (LOG), University of Lleida, Spain

Carlos Ansótegui ✉ 🆔
Logic & Optimization Group (LOG), University of Lleida, Spain

Josep M. Salvia ✉ 🆔
Logic & Optimization Group (LOG), University of Lleida, Spain

Eduard Torres ✉ 🆔
Logic & Optimization Group (LOG), University of Lleida, Spain

─── **Abstract** ───

In this paper, we describe how we can effectively exploit alternative parameter configurations to a MaxSAT solver. We describe how these configurations can be computed in the context of MaxSAT. In particular, we experimentally show how to easily combine configurations of a non-competitive solver to obtain a better solving approach.

1 Introduction

Since 2006, the MaxSAT Evaluation (MSE) [5] has been held annually with the primary objective of advancing MaxSAT technology and assessing its current state-of-the-art. The evaluation consists of multiple solvers being tested on various benchmarks across different evaluation tracks. This event has undeniably spurred the MaxSAT community to create more cutting-edge solvers and enhance their competitiveness.

It is not surprising that solver performance depends on several factors, including the *power* of the algorithm implemented by the solver, proper configuration of solver parameters to unleash its full potential, and implementation issues. Therefore, we must interpret the MaxSAT Evaluation ranking results carefully and derive conclusions according to the goal of our analysis. For example, a similar or *weaker* algorithm could outperform other approaches thanks to better implementation of data structures or a previous tuning process of its input parameters.

From an industrial point of view, we mainly care about obtaining an effective solving approach that is ready for deployment for a particular problem subject to available resources (computing power, environment restrictions, licenses available, etc). From a research point of view, we are more interested in identifying the potential of new solving approaches that lead to further promising research avenues.

Our aim is to satisfy both industrial and research perspectives by identifying the best possible solving approach that can be achieved from a single solver while adhering to certain restrictions. In particular, we treat the solver as a black box, meaning that we cannot access its source code, nor do we have any domain knowledge of the problem to be solved, meaning that we cannot utilize any specific structure feature.

Despite these constraints, our approach enables us to unleash the hidden potential of the solver and avoid incorrect rankings of better algorithms that have not been appropriately configured or restarted. Additionally, our study emphasizes the importance of being cautious when interpreting rankings based on the MaxSAT Evaluation, as we mentioned previously.

In this paper, we first show how to effectively configure MaxSAT solvers using Automatic Configuration (AC) tools (tuners), specifically GGA [4] and SMAC [14]. Then, we show that we can take advantage of not only the best configuration returned by the tuners but also a selection of the configurations seen by the tuner during the AC process. With these configurations, we can then build a simple portfolio that runs in parallel these configurations if enough computational resources are available.

We also demonstrate how to create a sequential portfolio that schedules the execution of different parametrizations of a single MaxSAT solver on a given number of cores within a specified timeout. This approach can be thought of as a restarting strategy, where a different configuration of the solver parameters is selected at each restart.

It is worth mentioning that all these approaches are agnostic of the structure of the instances. Otherwise, we should explore extending other approaches available in the literature such as ISAC++ [13].

Finally, we integrate all these building processes in the OptiLog framework [1]. With the new APIs, the user can provide an input MaxSAT solver and its parameters through the *BlackBox Module*, and OptiLog automatically generates a new solving approach for a given number of cores.

We conducted an extensive experimental investigation on the Weighted Incomplete track of the MaxSAT Evaluation 2022, with a particular focus on the highly configurable MaxSAT solver Loandra [6]. In this track, Loandra ranked sixth when restricted to a timeout of 60 seconds. Our approach involves the construction of parallel and sequential portfolios based solely on Loandra, which significantly improves its performance.

2   Preliminaries

MaxSAT is the optimization variant of the SAT decision problem. While for SAT the goal is to find an assignment to the Boolean variables (solution) that satisfies all the clauses in the input CNF formula, in MaxSAT we look for a solution that satisfies the maximum possible number of clauses. Since some of these clauses can be falsified we refer to them as *soft* clauses. Within the MaxSAT community, it is typical to reformulate the problem from a minimization perspective aiming to find a solution that falsifies the minimum possible number of soft clauses.

There are several variants of the MaxSAT problem. We can add weights to the soft clauses that represent the cost of falsifying the clause. In this case, we want to look for a solution that minimizes the aggregated cost of the *Weighted* soft clauses. Additionally, we can have *hard* clauses, i.e., clauses that cannot be falsified by the solution.

MaxSAT solvers have experimented a great success in the last decade. Among these solvers, we find complete (or exact) solvers and incomplete solvers. Complete solvers provide optimal solutions while incomplete solvers report solutions as good as possible, but are not

required to guarantee their optimality. These solvers can either refine a lower bound (lb) on the cost of the optimal solutions or an upper bound (ub), or both. In particular, incomplete solvers iteratively report (whenever possible) a better (smaller) upper bound on the optimal solution.

3 The MaxSAT Evaluation

The MaxSAT Evaluation 2022 was structured into three tracks: main track complete (unweighted and weighted variants), main track incomplete (unweighted and weighted variants), and the special incremental MaxSAT track. In this paper, we focus on the incomplete track for weighted MaxSAT instances with a timeout of 60 seconds.

The term *incomplete* refers to the type of MaxSAT solvers which are not required to be exact, i.e., they do not need to certify the optimum. Their goal is to report the best possible solution within a given timeout. The term *weighted* refers to the variant of MaxSAT instances. The weighted MaxSAT variant allows integer weights for the soft clauses plus the hard clauses.

We consider the timeout of 60 seconds useful for our study since it is a realistic scenario of industrial applications where we require a suboptimal solution in a short time window and because our automatic configuration process, given the computational resources we have available, can be restricted to two days (see Section 5).

The MaxSAT Evaluation 2022 incomplete (weighted) track involved 197 MaxSAT instances and 10 incomplete solvers: DT-HyWalk [18], noSAT-MaxSAT [15], NuWLS-c [7], Exact [8, 11], Loandra [6], Open-WBO-inc (two variants) [12], and TT-Open-WBO-Inc (three variants) [16].

Each solver s was ranked according to the scoring function $score(s)$ shown in Equation 1.

$$score(s) = \frac{\sum_{i=1}^{i=n} score(s,i)}{n} \tag{1}$$

Given a set of n instances, the $score(s)$ of a MaxSAT solver s is the average of the scores for each instance, computed by $score(s,i)$ in Equation 2.

$$score(s,i) = \frac{1 + \text{best-known ub for instance } i}{1 + \text{ub for } i \text{ found by } s} \tag{2}$$

The $score(s,i)$ function computes the ratio between the *best-known upper bound* of an instance i and the bound reported by the solver s on the same instance. Assuming that (*best-known ub*) $\leq ub$ (which is the case for the MaxSAT Evaluation), the computed value ranges between 0 and 1, where higher values correspond to better upper bounds.

The competition has some specific rules about what is and is not allowed in the implementation of the solvers. In particular, the solvers are not allowed to employ triggers to modify their behavior, which is deemed to be specific to particular instances. However, solvers can concatenate the usage of different solving techniques.

In the most recent competition, the MaxSAT solvers used a variety of strategies and solvers. Some of these solvers are outlined below, along with the various approaches they employ in order to find improved solutions.

1. **NuWLS-c**: This solver adopts two solvers, the NuWLS solver, which is an improvement of SATLike, and the integration of TT-Open-WBO-Inc.
2. **TT-Open-WBO-Inc**: This solver uses four different strategies, including SATLike for inprocessing, a modified version of Mrs. Beaver for unweighted instances, BMO-clustering for weighted instances, and Polosat, a SAT-based local search method. This MaxSAT solver has three different distributions:

- **(g)** Which incorporates the Glucose 4.1 SAT Solver.
- **(i)** Which incorporates the new Intel SAT Solver.
- **(is)** Which incorporates the new Intel SAT Solver and is tuned for short invocations.
3. **DT-HyWalk**: This solver employs three distinct strategies, including a direct call to a SatSolver, the SATLike solver for local search, and the use of another MaxSAT solver, TT-Open-WBO-Inc.
4. **Loandra**: This solver utilizes two core algorithms, namely a Core-Based algorithm and a Linear algorithm.

Table 1, column "MSE", shows the results of the MaxSAT Evaluation 2022 for the top six solvers at the incomplete weighted track with 60 seconds timeout. As we can see, the MaxSAT solver Loandra was not competitive within this category. In this paper, we propose an approach that is agnostic of the structure of instances and only allows the usage of alternative configurations of the same input MaxSAT solver. We experimentally show the goodness of our approach on the MaxSAT solver Loandra.

3.1   Reproducing the MaxSAT Evaluation for the Incomplete track

All of our executions of the MaxSAT solvers are run on a computation cluster composed of nodes with two AMD 7402 processors (each with 24 cores at 2.8 GHz) and 21 GB of RAM per core, managed by Sun Grid Engine (SGE). All the experiments are managed using the *Running Module* of the OptiLog framework.

Each execution is given 60s of CPU time and 32 GB of memory. As the memory requirements exceed the memory per core available, two slots are reserved and an affinity mask is set by SGE to restrict the execution to only one of the two cores. In contrast to the MaxSAT evaluation, each solver was evaluated with 50 different random seeds and we report results on the average score, and in some of the experiments, we also show the minimum and maximum scores, and the standard deviation.

In the course of developing our experiments, we detected two problems with some executions of the solvers: 1) some executions report a bound that does not correspond to the real cost of the solution reported, and 2) some executions report a solution that does not satisfy the *hard* clauses.

To address these issues we conduct a validation step executed after the solver exhausts the 60s of CPU time. In particular:

For 1), we trust the cost we compute from the solution reported, ignoring the bound reported by the solver.

For 2), we consider the solver was not able to find any solution at all.

This validation step is also conducted during the automatic configuration process when we evaluate a particular configuration of the solver on a given instance (see Section 5).

The score for each solver is computed using the MaxSAT evaluation rules. In particular, it is important to define which is the set of best-known upper bounds that we use to compute the score. Table 1 shows in column "MSE 2022", the scores reported in the MaxSAT Evaluation 2022.

The rest of the columns present the results of the experimentation we conducted (in our cluster) using different sets of *best-known upper bounds*. "VBS_b" uses the upper bounds found by the Virtual Best Solver of the solvers we executed, "MSE_b" uses the set of best-known upper bounds provided by the MaxSAT Evaluation, and "$LRUNS_b$" (Long Runs) uses a set of new *best-known upper bounds* we computed by running Loandra and NuWLS-c (both with the default parameters) with a timeout of 12 hours. We recall that the score presented

is the average score on 50 seeds in contrast to the MSE results where only 1 seed is used. As we can observe in Table 1, the relative ranking of the solvers is preserved although we can observe variations in the scores reported. We will use in the rest of the paper the best bounds from $MSE_b + VBS_b + LRUNS_b$.

Table 1 Results of the MaxSAT Evaluation 2022 on the incomplete weighted track (60 seconds timeout) and reproduction of the Evaluation in our system with different sets of Best-Known Upper Bounds.

|	MSE 2022	MSE 2022 on our system		
Best-known UBs	MSE_b	VBS_b	$VBS_b + MSE_b$	$VBS_b + MSE_b + LRUNS_b$
Solvers				
NuWLS-c	0.759	0.7831	0.7590	0.7524
DT-Hywalk	0.732	0.7625	0.7414	0.7351
TT-Open-WBO-inc (g)	0.728	0.7412	0.7221	0.7164
TT-Open-WBO-inc (is)	0.726	0.7354	0.7201	0.7141
TT-Open-WBO-inc (i)	0.720	0.7354	0.7178	0.7118
Loandra	0.693	0.7107	0.7003	0.6953

4   Automatic Configurators (AC)

In this section, we review the Automatic Configuration Problem and two state-of-the-art automatic configuration algorithms or tuners.

4.1   The Automatic Configuration Problem

Given a target algorithm A with parameters $\{p_1, \ldots, p_n\}$ of domain $d(p_i)$. We define the parameter space Θ of A as the subset $d(p_1) \times \ldots \times d(p_n)$ of *valid* parameter combinations. Depending on the parameter, $d(p_i)$ can be categorical, a discrete domain of fixed values with no predefined order, or numerical, which represent integer or real values. Then, we define the *Automatic Algorithm Configuration* (AAC) problem as the optimization problem that consists of exploring Θ to find a configuration $\theta \in \Theta$ of A, which given a set of problem instances Π, minimizes a cost metric $\hat{c} : \Theta \times \Pi \to \mathbb{R}$, without exceeding a configuration budget B.

It is common for A to be a black box (target algorithm), meaning it accepts some inputs (the parameters) and provides some output (e.g., $\hat{c}$), but we cannot see its internal functionality. This allows AAC to generalize to any type of algorithm but makes it more challenging for algorithm tuners since they cannot use A to infer additional information about Θ. In practice, A is implemented as a binary file that outputs its results in a format suitable for its domain but may not be suitable for the AAC tool. Moreover, it may be necessary to limit the resources that A can use to solve an instance, such as memory or CPU time. The standard way of addressing these issues in AAC tools is for the user to replace A with a wrapper script that handles these and any other necessary aspects. Figure 1 describes the automatic configuration process where the tuner is a solver for the AAC problem.
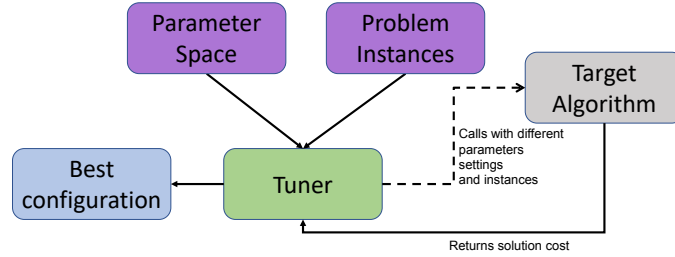
■ **Figure 1** Visualization of the Automatic Configuration process.

4.2 The GGA Automatic Configurator

The Gender-Based Genetic Automatic Algorithm Configuration (GGA) is a genetic algorithm that was introduced in [4] to search for high-quality configurations. It was one of the pioneering algorithms that supported continuous parameters and introduced the novel concept of *gender* to apply diverse selection pressures to the population's individuals.

■ **Algorithm 1** GGA.

Input: Target Algorithm A, Parameter Space Θ, Instances Π, Performance Metric $\hat{c}$, # MiniTournaments N, Configuration Budget B

1: **function** GGA($A, \Theta, \Pi, \hat{c}, N, B$)
2:     pop $\leftarrow$ initPopulation(Θ)
3:     $j = 0$
4:     **while** B not exhausted **and** threshold not achieved **do**
5:         $j = j + 1$
6:         $\Pi_j \leftarrow$ selectInstances(Π, j)
7:         $<w_1, ..., w_N> \leftarrow$ runMiniTournaments(A, pop.comp, Π_j, $\hat{c}$, pop.comp/N)
8:         offspring $\leftarrow$ applyCrossoverAndMutate(pop.noncomp, $<w_1, ..., w_N>$, Θ)
9:         pop $\leftarrow$ agingAndDeath(w_1, pop) $\cup$ offspring
    return w_1

Algorithm 1 shows the pseudocode of the GGA algorithm, which takes as input the target algorithm A, its parameter space Θ, a set of training instances Π, a performance metric $\hat{c}$ to optimize (e.g., time, accuracy, quality within a fixed timeout, etc), the number N of GGA mini-tournaments (which will be explained shortly), and a configuration time budget B.

GGA starts by initializing a *population* (*pop*) of configurations (named *genomes*) as a subset of Θ in line 2. This population is partitioned into a *competitive* group (*pop.comp*, which is directly evaluated on the target algorithm) and *non-competitive* group (*pop.noncomp*, which simply acts as a source of diversity).

The algorithm proceeds in a main loop that finishes when GGA reaches the configuration budget B or a threshold on the performance (line 4). At each iteration (which we call *generation*), GGA selects a subset of the instances Π_j to evaluate the genomes in line 6[1]. Then, in line 7, GGA evaluates the competitive genomes of the population over the selected instances Π_j using a parallel racing scheme called *mini-tournament*. This procedure returns

---

[1] There are different policies that can be applied to select the instances at each generation, see [4].

a set of N winners, $<w_1, ..., w_N>$, which will be the only competitive genomes that will generate new offspring in this generation (line 8). Finally, GGA applies an ageing policy in line 9 that is used to prevent population growth. The only exception is the overall best competitive genome (w_1), which survives as long as it performs better than the other mini-tournament winners. At the end of the main loop, GGA returns the best competitive genome w_1 of the last generation.

For more details on the GGA algorithm, we refer the reader to [4].

4.3 The SMAC Automatic Configurator

Sequential Model-Based Algorithm Configuration (SMAC) is an automatic configuration algorithm based on Bayesian optimization [10, 14]. In Bayesian optimization, we use a few evaluations of the target algorithm to train a *surrogate model* that predicts the performance of the algorithm for a given configuration. This fast-to-evaluate surrogate model is used to search for promising new configurations that will be executed on the training instances.

■ **Algorithm 2** SMAC.

Input: Target Algorithm A, Parameter Space Θ, Instances Π, Performance Metric $\hat{c}$, Configuration Budget B

1: **function** SMAC($A, \Theta, \Pi, \hat{c}, B$)
2:    $[R, \theta_{inc}] \leftarrow$ initialize(Θ, Π)
3:    **while** B not exhausted **do**
4:        $[M, t_{fit}] \leftarrow$ fitModel(R)
5:        $[\vec{\Theta}_{new}, t_{select}] \leftarrow$ selectConfigurations(M, θ_{inc}, Θ)
6:        $[R, \theta_{inc}] \leftarrow$ intensify($A, \vec{\Theta}_{new}, \theta_{inc}, R, \Pi, \hat{c}$)
        return θ_{inc}

Algorithm 2 shows the pseudocode of SMAC. This algorithm receives as input the target algorithm A, its parameter space Θ, a set of training instances Π, a performance metric $\hat{c}$ to optimize and a configuration time budget B. First, SMAC initializes a *best candidate* configuration θ_{inc} and the history of conducted evaluations of different *(configuration, instance)* pairs R (which might be empty) in line 2.

As in GGA (see Section 4.2), SMAC has a main loop defined in line 3 that proceeds until the configuration budget B is reached. At each iteration, it fits a surrogate model M using the information in R in line 4. Then, it uses M to select a new set of promising candidate configurations $\vec{\Theta}_{new}$ in line 5. Finally, it evaluates $\vec{\Theta}_{new}$ and θ_{inc} on instances from Π to determine the next best candidate θ_{inc}, according to $\hat{c}$ in line 6. Similar to the GGA algorithm, SMAC returns the best candidate configuration θ_{inc} .

4.4 Support for Tuning into the OptiLog framework

In this section, we present an excerpt of the code that uses the OptiLog framework to generate the configuration environment (from now on Tuning Scenario) of the solver Loandra for GGA and SMAC tuners.

```
1 # example_ac.py
2
3 from optilog.blackbox import *
4 from optilog.running import ParsingInfo
5 from optilog.tuning import *
6
```

```
7  class LoandraBB(SystemBlackBox):
8      config = {
9          "weight−strategy": Int(0, 2, default=2),
10         "preprocess": Bool(default=True),
11         (...)
12     }
13     (...)
```

◾ **Listing 1** Sample code to wrap the solver Loandra into an OptiLog BlackBox.

Listing 1 defines a custom BlackBox class named LoandraBB that inherits from System-BlackBox. This class represents the binary that we want to optimize. The `config` dictionary defines the parameters of this binary that can be tuned by the optimization algorithm. We show the parameters "weight-strategy" and "preprocess", with their respective types and default values. In particular, we need 18 lines of code to wrap Loandra, with 40 additional lines defining the parameters.

```
1  # scenario_gga.py
2
3  from optilog.tuning.configurators import *
4  from example_ac import LoandraBB
5
6  if __name__ == "__main__":
7      configurator = GGAConfigurator(
8          LoandraBB(),
9          input_data="/path/to/instances/*",
10         ...
11     )
12     configurator.generate_scenario("./scenario")
```

◾ **Listing 2** Sample code to create a Tuning Scenario for the solver defined in Listing 1.

Listing 2 is a definition of a Tuning Scenario for the solver Loandra. It imports the custom `LoandraBB` class defined in Listing 1, and sets up a tuner (in this case GGA) to optimize the parameters of `LoandraBB`. The `input_data` parameter specifies the path to the instances used during the optimization process. Lastly, the `generate_scenario` method is called with the desired output path for the scenario that is being created.

Similar code to Listing 2 could be used to define a Tuning Scenario to be used with the SMAC AC tool, as OptiLog supports both GGA and SMAC.

5   Configuring MaxSAT Solvers

Although the AC tools (tuners) presented in the previous section have also parameters that impact the effectiveness of the configuration process, *tuning the tuner* is out of reach in this paper and we focus on providing a good cost function to be used during the tuning process.

Ideally, we would use the $score(s, i)$ function from the MaxSAT Evaluation (Equation 2). Notice though that in the MaxSAT Evaluation we are trying to maximize this scoring function, whereas tuners minimize a cost (see Section 4). Therefore, we have to convert the *score* function to a *cost* function. Additionally, it is not guaranteed that the bounds found are equal or worse than the previously *best-known upper bounds* (see Section 3), and we cannot update the *best-known upper bounds* sets during the tuning process (otherwise previous results computed in the same tuning process would not be comparable).

We define the $cost_{ac}(s, i)$ function, shown in Equation 3, as follows. First, we split the function in two cases: 1) the reported bound is worse (or equal) than the previous *best-known upper bound*, and 2) the bound reported is better.

For 1), we compute $1 - score(s, i)$ to obtain a value in the range $[0, 1)$, where better bounds are closer to 0.

For 2), notice that we are breaking the assumption $(best\text{-}known\ ub) \le ub$, which may lead to unbounded values that tend to ∞. To restrict the values to the range $(-1, 0)$, we use the inverse of the $score(s, i)$ function, and then subtract 1.

The $cost_{ac}(s, i)$ function returns values between $(-1, 1)$, where better bounds correspond to values closer to -1.

$$cost_{ac}(s, i) = \begin{cases} 1 - score(s, i), & \text{if ub for } i \text{ found by } s \ge \text{best-known ub for } i \\ \frac{1}{score(s,i)} - 1, & \text{otherwise} \end{cases} \tag{3}$$

As stated in Section 3.1, some executions might report a bound that does not match the reported solution. Thus, we integrate a validation step (see Figure 2) that certifies the real cost of the solution returned by the solver and reports it to the tuner.

Regarding the tuning environment, all experiments are conducted on the same computation cluster. Each tuning process is given a wall-time tuning budget of 48 hours, a memory limit of 32G per worker, and is allowed to use up to 50 parallel workers unless otherwise specified. Each configuration instance is given a CPU time limit of 60 seconds for the solver, and then a validation step is executed. As training instances for the tuning process, we will use the 197 instances from the MaxSAT Evaluation 2021 (incomplete weighted track, 60 seconds) and we will test the best configuration returned by GGA and SMAC (see Section 4) on the 151 instances from the MaxSAT Evaluation 2022.

GGA allows for the selection of how many instances are used in each generation. Incrementally increasing the training set across generations till including the whole set of available instances is often recommended, as it facilitates discarding bad configurations with less effort, therefore more generations can be reached within the tuning budget. However, as discussed in Section 6, prioritizing the evaluation of more configurations on the whole training set within the same tuning budget may be preferable over having more generations. GGA also can preserve a set of *elite* configurations that are run at every generation. We define as an *elite* the default configuration of Loandra. Finally, we use the PyDGGA [2] (version 1.7.0) distribution of GGA which has support for distributed execution. The following non-default parameters were used for GGA: cost tolerance set to 0, population set to 100, generations set to 300, and minimum generations set to 50.

Regarding SMAC, although it can be executed in parallel, it does not report an overall winner in contrast to GGA. Instead, it reports as many winners as computation cores were used since it basically runs several sequential SMACs in parallel. Thus, after SMAC completes, we have to take all the winners from each SMAC sequential execution, which may not have been evaluated on all training instances, and perform the missing evaluations. Then, the winner with the best performance on the training set is selected to be the overall winner. We use the SMAC3 [14] (version 1.4.0) implementation of the algorithm.

As we have described earlier, the cost function used during the tuning process is not *strictly* the minimization version of the score we maximize according to the MSE 2022 (see Section 3). Therefore, one may argue that it would be better to return a winner for the training set with respect to the score function computed by the MSE. This is easy to do if the tuner provides the logs of each evaluation so, in the case of incomplete MaxSAT solvers, we can retrieve the best bound found by the solver on a given instance.

Therefore, we add a *selection phase* (see Figure 2) after the tuning phase that recomputes the scores (according to the MSE) of the configurations traversed by the tool during the tuning process.
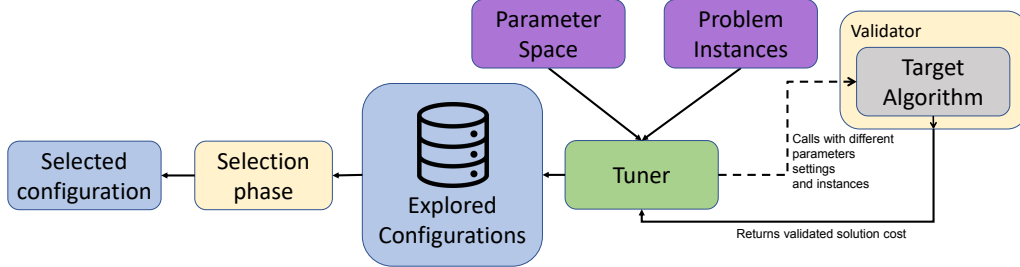
■ **Figure 2** Visualization of the Automatic Configuration process extended with the validator and a selection phase over the explored configurations.

In particular, for SMAC, we compute the MSE score of the 50 winners reported by the tuner on the training set and select the one with the highest score. Even though we have access to the logs of the evaluations of SMAC and could use those scores to select the winning configuration, we need to make sure that all the configurations are evaluated with all the instances.

For GGA, we order the configurations first by their ranking in a generation (according to the cost function in Equation 5), and within the same rank, we order by the most recent generation. Then, we select the first 50 distinct configurations. We look into their logs, recompute their MSE score according to Section 3, and report the winner[2].

■ **Table 2** Comparison using GGA and SMAC to tune the Loandra solver (using $VBS_b + MSE_b + LRUNS_b$ bounds).

	Mean	Median	Min	Max	Std
NuWLS-c	0.7524	0.7522	0.7484	0.7560	0.0017
Loa (GGA, all-i)	0.7393	0.7391	0.7313	0.7475	0.0037
Loa (GGA, incremental)	0.7353	0.7354	0.7275	0.7433	0.0038
DT-Hywalk	0.7351	0.7355	0.7288	0.7415	0.0030
Loa (SMAC)	0.7237	0.7234	0.7149	0.7355	0.0048
TT-Open-WBO-inc (g)	0.7164	0.7165	0.7128	0.7194	0.0015
TT-Open-WBO-inc (i)	0.7141	0.7142	0.7093	0.7188	0.0020
TT-Open-WBO-inc (is)	0.7118	0.7117	0.7098	0.7145	0.0008
Loandra	0.6953	0.6957	0.6872	0.7036	0.0037

Table 2 shows the result of the best configurations provided by GGA using all the instances from the first generation ("*Loa (GGA, all-i)*"), or adding them incrementally at each step ("*Loa (GGA, incremental)*") and the best configuration provided by SMAC (see

---

[2] In our experiments, the winner reported by GGA was the same configuration as the best one found in the *selection phase*.

"*Loa (SMAC)*") after the additional selections process described in the paragraph above. For the incremental approach of GGA, we use 20% of instances at the first generation and instruct GGA to use all the instances on generation 25. Those values were selected based on preliminary experiments taking into account the number of generations that GGA can do in the given time. It is clear by the results that the usage of all the instances from the beginning benefits GGA, allowing it to lift Loandra from the sixth position to the second one. In the next section, we will focus on the variant of GGA "*Loa (GGA, all-i)*".

We also ran the variants "*Loa (GGA, all-i)*" and "*Loandra*" on another set of benchmarks, the MSE 2020 instances. The default parameters variant achieves a score of 0.755, and the tuned version a score of 0.777.

6 Exploiting Configurations *Discarded* by the Tuner

As it has been shown in the literature [9, 17], from the most pragmatic point of view, we can obtain an efficient parallel approach by just running the same non-deterministic solver with different seeds in parallel, or we can also run in parallel different configurations of the same solver.

In case resources are limited, we can also schedule the execution of different configurations of the same solver. In this section, we concrete and study these different approaches. We use OptiLog [1] to generate all the portfolios, as we explain in Section 6.3.

6.1 Parallel Portfolios of seeds and configurations

As we have already explained, tuners report the best configuration they have found. However, many other *potentially* good configurations are also explored and discarded during the automatic configuration process with respect to their performance on the particular training set. These configurations may exhibit good performance in different kinds of instances. As observed in [3] on SAT benchmarks, superior performance can be achieved by combining these complementary configurations.

The first approach we explored is the parallel execution of N different random seeds over a given MaxSAT solver. This approach can be applied to both the *default* MaxSAT solver and the best configuration obtained in the tuner.

Another approach is to extract N configurations of a MaxSAT solver from the ones traversed by the tuner and execute them in parallel. There are many strategies that we could follow to extract these configurations from the tuner. In particular, we use the set of configurations considered during the *selection phase* (after the *tuning phase*) process as explained in Section 5. Notice that we do not analyze any structure of the instances and we only incorporate configurations of the same solver.

Table 3 shows the results of the parallel portfolios that we explained. We tested parallel portfolios with 25, 30, 35, 40, 45, and 50 parallel executions. Each row shows the results of a parallel portfolio (rows marked with *(Seeds)* refer to a parallel portfolio of seeds, whereas the row marked with *(Configs)* refer to a parallel portfolio of configurations). We show the score as computed in the MaxSAT evaluation using the $VBS_b + MSE_b + LRUNS_b$ upper bounds and the rank of each portfolio with respect to the others.

As we can observe, the portfolio over different seeds for the default Loandra ("*(Seeds) Loandra*" in Table 3) is not competitive while the portfolio of different seeds for the best configuration of Loandra computed by GGA (column "*(Seeds) Loa (GGA, all-i)*") already outperforms NuWLS-c. Additionally, a portfolio of the best configurations provided by the selection phase (column "*(Configs) Loa (GGA, all-i)*") systematically outperforms the rest of the approaches. These observations hold almost for any number of parallel executions.

■ **Table 3** Score and rank (#) for each parallel portfolio, given N parallel processes (using $VBS_b + MSE_b + LRUNS_b$ bounds).

N		25		30		35		40		45		50
	#	score	#	score	#	score	#	score	#	score	#	score
(Configs) Loa (GGA, all-i)	1	0.813592	1	0.818663	1	0.823986	1	0.825208	1	0.826448	1	0.827105
(Seeds) Loa (GGA, all-i)	2	0.806889	2	0.808078	2	0.809922	2	0.812589	2	0.813007	2	0.815263
(Seeds) NuWLS-c	3	0.768409	3	0.769302	3	0.769785	3	0.770293	3	0.771219	3	0.771263
(Seeds) DT-Hywalk	4	0.759271	4	0.764688	4	0.765736	4	0.765764	4	0.765862	4	0.766397
(Seeds) TT-Open-WBO-inc (g)	5	0.728901	5	0.728903	5	0.729766	5	0.729902	5	0.730165	5	0.730235
(Seeds) TT-Open-WBO-inc (i)	6	0.725971	6	0.726171	6	0.726253	6	0.726315	6	0.726510	6	0.727468
(Seeds) Loandra	8	0.717788	8	0.722663	7	0.723954	7	0.723996	7	0.724521	7	0.724648
(Seeds) TT-Open-WBO-inc (is)	7	0.722672	7	0.722704	8	0.722918	8	0.723174	8	0.723324	8	0.723489

6.2 Sequential Portfolios of configurations

In some settings, we will not have enough resources to run a parallel portfolio as described in Section 6.1. Potentially, we can have just one computation core available. In this case, we can schedule the sequential execution of different configurations of Loandra within the given timeout.

Let us describe how we construct this *sequential* portfolio. We assume we have a sequence of solvers (or configurations of a solver) (S) that iteratively report better solutions, a time budget (TO), and a maximum time budget a solver can exhaust between two consecutive reported solutions ($MTBS$). The solvers are executed according to their order in the sequence until the time consumed globally by all the solvers exceeds TO.

Each solver is run as follows: first, we wait for the first solution reported by the solver. Once this first solution is reported, we start a timer of $MTBS$ seconds. If the solver reports a new solution before this timer expires, we reset the timer and wait for a new solution. This is repeated until the solver is unable to report a new solution before the timer is consumed. At that point in time, the solver is stopped and the next one in the ordered list of solvers is executed. Note that, at any point in this process, a solver can also be stopped if the global time budget of TO seconds gets exhausted. A special case is the last solver of the sequence, which is allowed to run until the time budget expires (i.e. it is not stopped even if it took more than $MTBS$ seconds to find a new solution). Obviously, we keep track of the best overall solution seen so far.

To identify which sequence of solvers S and $MTBS$ value the portfolio should use, we carry out a simulation of sequential portfolios with the configurations provided by the *selection phase* (see Section 5) and their respective logs on the training instances computed during the *tuning phase*. In particular, we explore all sequences of up to size 3 and $MTBS$ values of $\{2, 3, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$ seconds. Once we identify the best *virtual* sequential portfolio for the training instances, we simulate again the execution of this *virtual* sequential portfolio on the test set. In Table 4 we present the results of this simulation.

To implement this *virtual* sequential portfolio we would need to take into account an additional thread that keeps track of the evolution of the solvers in the sequence, which may decrease the overall performance. Therefore, we see this *virtual* sequential portfolio as a restarting policy that MaxSAT developers could integrate into their solvers, with the added benefit that they may be able to reuse information computed by each solver in the sequence.

Table 4 shows the results of the *virtual* sequential portfolios (rows prefixed with "Virtual portfolio"), compared to the results that obtained the solvers from the competition with the default parameters, and with the best approach obtained using a tuner ("*Loa (GGA, all-i)*"). As in the MSE we run each solver with the same seed, except for NuWLS-c for which we also

Table 4 Score of the *virtual* sequential portfolio compared with the single-execution approach (using $VBS_b + MSE_b + LRUNS_b$ bounds).

	Score
Virtual sequential portfolio (N=2) - Loa (gga, all-i)	0.7642
Virtual sequential portfolio (N=3) - Loa (gga, all-i)	0.7642
NuWLS-c (max score on 50 seeds)	0.7560
NuWLS-c	0.7554
Loa (gga, all-i)	0.7513
DT-Hywalk	0.7432
TT-Open-WBO-inc (g)	0.7214
TT-Open-WBO-inc (i)	0.7180
TT-Open-WBO-inc (is)	0.7180
Loandra	0.6965

report on the best score value from 50 seeds. The N value shown in the *virtual* sequential portfolios rows indicates the length of the solvers' sequence. The portfolios are built on top of the configurations obtained after the selection phase with ("*Loa (GGA, all-i)*").

We notice that *virtual* sequential portfolios do perform better than NuWLS-c, and a selection of two configurations suffices to that end. Interestingly, if we build the *virtual* sequential portfolio on the test instances from the MSE 2022, then we get a better portfolio using three configurations that achieves a score of 0.7689, however, we cannot predict this portfolio based on the analysis we perform on the training instances from the MSE 2021.

Additionally, we conducted experiments to analyze the potential of combining solvers. In particular, we used the solver TT-Open-WBO[3]. We tuned this solver, and generated a *virtual* sequential portfolio combining the best configurations of TT-Open-WBO and Loandra. The *virtual* sequential portfolio (based on the analysis of the performance on the MSE 2021) obtained a score of 0.779 on the MSE 2022, which is the best score obtained for single-core evaluations. In comparison, the individual performance of Loandra and TT-Open-WBO after tuning is 0.747 and 0.751 respectively.

6.3 OptiLog Portfolio Generator

To facilitate the creation of the parallel and *virtual* sequential portfolios, we added support to compute them using the OptiLog framework.

```python
from optilog.portfolio import get_parallel_portfolio

get_parallel_portfolio(
    gga_scenario='./tuning-scenario',
    n_solvers=10,
    save_to='./parallel-portfolio'
)
```

Listing 3 Computing a parallel portfolio with OptiLog.

---

[3] A version provided by the author of the solver with the parameters exposed.

Listing 3 shows how we can generate and save a parallel portfolio with OptiLog. This portfolio is built by selecting N configurations as explained in Section 5, thus requiring a Tuning Scenario (generated with OptiLog as seen in Section 6.3). The function `get_parallel_portfolio` receives as parameters the Tuning Scenario that contains the results of the tuning process (`gga_scenario`), the number of solvers that will compose the parallel portfolio (`n_solvers`), and the directory where the scripts to launch each individual solver that composes the portfolio will be saved (`save_to`).

```
1 from optilog.portfolio import get_sequential_portfolio
2
3 get_sequential_portfolio(
4     path_scenario="./running-scenario",
5     n_solvers=2,
6     solution_regex=r"^o\s(\d+)",
7     save_to="./sequential-portfolio",
8     score_fn=maxsat_score_fn,
9     max_time_between_solutions=[5, 15, 25, 35]
10 )
```

◾ **Listing 4** Computing a sequential portfolio with OptiLog.

Listing 4 shows how we are generating a sequential portfolio with the results of a Running Scenario. Note that to generate the *virtual* sequential portfolio we require the full trace of the solvers (in particular for the incomplete MaxSAT case, we need the evolution of the best bound over time), so we cannot build it from a Tuning Scenario directly. The parameters `gga_scenario`, `save_to`, and `n_solvers` mean the same as in the function to compute a parallel portfolio. Additionally, we have to specify the following parameters: `score_fn` is used to transform the lines matched by `solution_regex` to a score that the portfolio will try to maximize (in this example the score function is $score(s)$ defined in Section 3), and `max_time_between_solutions` contains the possible values that the portfolio can choose from when selecting the parameter $MTBS$.

7   Conclusions

Given a target solver, we have presented an approach to easily generate a potentially much better solving approach. To this end, we exploit a set of alternative configurations of the same target solver coming from the residues of a tuning process. It is important to notice that we do not exploit any structure feature of the input problem or instance since in some domains these features are not easy to compute. In particular, we have shown how from a MaxSAT solver with a low ranking in one of the tracks of the MSE 2022 we can obtain a more competitive approach.

Our sequential portfolio generation approach can be seen as a first attempt to come up with effective restarting policies for MaxSAT solvers, something that has not been studied in depth in the literature.

Finally, the approach described has been integrated into the OptiLog framework avoiding the tedious process of setting up tuning environments and generating portfolios. Moreover, the API is general enough to be applied not only to MaxSAT solvers but to other solving approaches.

References

1 Josep Alòs, Carlos Ansótegui, Josep M. Salvia, and Eduard Torres. OptiLog V2: Model, Solve, Tune and Run. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISSN: 1868-8969. `doi:10.4230/LIPIcs.SAT.2022.25`.

2 Carlos Ansótegui, Josep Pon, and Meinolf Sellmann. Boosting evolutionary algorithm configuration. *Annals of Mathematics and Artificial Intelligence*, 2021. `doi:10.1007/s10472-020-09726-y`.

3 Carlos Ansótegui, Josep Pon, and Meinolf Sellmann. Boosting evolutionary algorithm configuration. *Annals of Mathematics and Artificial Intelligence*, 90(7-9):715–734, 2022. Publisher: Springer.

4 Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A Gender-based Genetic Algorithm for the Automatic Configuration of Algorithms. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, CP'09, pages 142–157. Springer-Verlag, 2009. tex.acmid: 1789011 tex.numpages: 16 tex.year: 2009 event-place: Lisbon, Portugal. URL: `http://dl.acm.org/citation.cfm?id=1788994.1789011`.

5 Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, Ruben Martins, and Andreas Niskanen. MaxSAT Evaluation 2022 : Solver and Benchmark Descriptions. *Department of Computer Science Series of Publications B*, B-2022-2, 2022. Accepted: 2022-08-25T10:09:01Z Publisher: Department of Computer Science, University of Helsinki. URL: `https://helda.helsinki.fi/handle/10138/347396`.

6 Jeremias Berg, Emir Demirovic, and Peter Stuckey. Core-boosted linear search for incomplete maxsat. *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 16th International Conference, CPAIOR 2019*, 2019.

7 Yi Chu, Shaowei Cai, Zhendong Lei, and Xiang He. Nuwls-c: Solver description. *MaxSAT Evaluation 2022*, page 28, 2022.

8 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 1291–1299, 2018. URL: `https://www.ijcai.org/Proceedings/2018/180`.

9 Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *JSAT*, 6:245–262, June 2009. `doi:10.3233/SAT190070`.

10 Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration (extended version). *International Conference on Learning and Intelligent Optimization*, 2011.

11 Jo Devriendt. Exact Solver Repository, April 2023. URL: `https://gitlab.com/JoD/exact`.

12 Saurabh Joshi, Prateek Kumar, Sukrut Rao, and Ruben Martins. Open-wbo-inc: Approximation strategies for incomplete weighted maxsat. *J. Satisf. Boolean Model. Comput.*, 11(1):73–97, 2019. `doi:10.3233/SAT190118`.

13 Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC – Instance-Specific Algorithm Configuration. In *ECAI 2010*, pages 751–756. IOS Press, 2010. `doi:10.3233/978-1-60750-606-5-751`.

14 Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. In *ArXiv: 2109.09831*, 2021. URL: `https://arxiv.org/abs/2109.09831`.

15 Ole Lübke and Sibylle Schupp. nosat-maxsat. In *MaxSAT Evaluation 2022*, pages 29–30. Department of Computer Science, University of Helsinki, 2022.

16 Alexander Nadel. Polarity and Variable Selection Heuristics for SAT-Based Anytime MaxSAT: System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, 12(1):17–22, September 2020. `doi:10.3233/SAT-200126`.

17   Olivier Roussel. Description of ppfolio 2012. In *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, 2012.

18   Jiongzhi Zheng, Kun He, Zhuo Chen, Jianrong Zhou, and Chu-Min Li. Decision tree based hybrid walking strategies. *MaxSAT Evaluation 2022*, page 24, 2022.