Universiteit Leiden
The Netherlands

Sampling strategies in automated algorithm configuration

Anastacio, M.I.A.

2

Automated Algorithm Configuration

In this chapter, we define the automated algorithm configuration (AAC) problem, its applications and challenges, particularly when optimising the running time of solvers for NP-hard problems. We list existing state-of-the-art configurators and delve into their inner workings to gain a deeper understanding of how they address these challenges during their search procedure. We then introduce the set of configuration scenarios and benchmarks that have been used in the literature and will be used throughout the thesis. For each of them, we attempt to define characteristics that might impact the performance of configurators. An empirical evaluation of configurator performance is presented in Chapter 3.

2.1 Algorithm configuration problem

Many state-of-the-art algorithms, especially solvers for NP-hard problems, come with parameters that enable users to fine-tune the inner workings of the solver to the specific problem instances they are trying to solve. This tuning has traditionally been conducted manually or through simple search procedures, such as random search, which remains the approach in many fields despite the availability of tools to automate parameter tuning. The question of finding which parameter values should be used for an algorithm to perform well on a set of problem instances is known as the *automated algorithm configuration problem*.

2.1.1 Problem definition

The AAC problem can be defined as follows (see *e.g.* Hoos, 2012a).

Definition 2.1. Given:

- a target algorithm A with k parameters $p_1, p_2, ..., p_k$, a domain $\mathcal{D}_j$ of possible values and a default value $D_j \in \mathcal{D}_j$ for each parameter p_j;
- a configuration space Ω, containing all valid combinations of parameter values of A;
- a set of problem instances $\mathcal{I}$;
- a performance metric $\mathcal{M}$ that measures the performance on $\mathcal{I}$ of target algorithm A configured according to $\omega \in \Omega$;

find $\omega^* \in \Omega$ that optimises the performance of A on instance set $\mathcal{I}$, according to metric $\mathcal{M}$.

Note that typically the running time of the target algorithm is limited by an upper bound defined by means of a cutoff time T_{cut}. This limit has an impact on any attempt to model the running time, as a saturation phenomenon occurs at the upper bound. Instead of following the underlying distribution of running time, data gathered with such a cutoff time will be set to T_{cut} for any algorithm that needs more time. Some configuration approaches will implement methods to mitigate this impact, such as SMAC proposed by Hutter et al. (2011b) (described in Section 2.2.2).

2.1.2 Challenges

The AAC problem comprises three main elements, each coming with its own challenges.

Target algorithm

The target algorithm is typically treated as a black-box algorithm with complex inner workings. Although some methods tried to open the black box (see *e.g.* Adriaensen and Nowé, 2016; Pulatov et al., 2022), these are still preliminary and, to the best of our knowledge, there exists no configurator that integrates such methods natively. In this work, we are particularly interested in solvers for NP-hard problems. Such algorithms present two significant challenges. First, their typical running time varies widely depending on the specific problem instance they solve, especially for solvers of NP-hard problems and the longer evaluations can be very costly. Thus, the configuration process can benefit from mechanisms to reduce the number of evaluations. Second, they do not typically come with a notion of anytime performance, preventing us from using methods typically used for machine learning algorithms, such as successive halving Jamieson and Talwalkar (2016) or learning curve-based performance prediction Domhan et al. (2015).

Configuration space

Each parameter comes with a domain of possible values. This domain can be of different types: categorical, ordinal, or numerical. Categorical parameters have an unordered finite set of possible values and are often used to select between several heuristic components or mechanisms. Numerical parameters are real- or integer-valued and are often used to calibrate heuristic mechanisms or components. Parameters can also conditionally depend on each other, such that one is active only when another takes a specific value. For example, consider a Boolean parameter that activates a mechanism, which is itself adjusted using a numerical parameter. While in some scenarios $\Omega = \mathcal{D}_1 \times \ldots \times \mathcal{D}_k$, these conditional relationships induce complex shapes in the resulting configuration spaces, as some parts are accessible only under the condition that other parameters are set to a specific value. Despite those challenges, Pushak and Hoos (2018) demonstrated that the performance landscapes (*i.e.* how the performance of the target algorithm varies depending on the variations in its configuration) tend rather benign, which can facilitate accurate prediction of the performance of the algorithm. Chapter 4 will delve more into the configuration space.

Instance set

The instance set comprises a set of instances that are considered representative of the given problem. Specifically, the instances in this set are expected to be drawn from

the same underlying distribution, such that the configuration obtained based on a part of this set performs similarly well on the whole set. Indeed, the AAC problem aims to find a well-performing configuration for a specific type of problem instances, rather than a configuration that performs well on any instance. The set can contain instances that are more or less difficult, and thus take more or less time to solve. This affects the configurator in many ways. Indeed, long running times mean that fewer evaluations can be performed within a given time budget. To avoid the configuration process from stalling on an unsolvable or extremely costly instance, each algorithm run is stopped after a cutoff time T_{cut}. However, if too many instances cannot be solved by the solver within this time, there is very little to be learnt from the runs. We look into this in more detail in Chapters 5 and 6.

2.2 Configuration approaches

In recent years, much work has been done on AAC, resulting in several general-purpose automatic algorithm *configurators*, based on different approaches. Each requires and includes three key mechanisms: a method to generate a challenger configuration from Ω, a method to estimate the performance of a configuration on the instances from $\mathcal{I}$, and a method to decide which configuration(s) to keep for the next iteration of the optimisation process.

2.2.1 Scope

Most configurators have limitations, strengths and weaknesses that are based on their underlying search algorithm and the configuration scenarios for which they were designed. In this work, we are interested in configuring algorithms that support a large number of parameters of any type (numerical, Boolean, or categorical) with possible conditional dependencies between them. We thus do not consider configurators that do not handle categorical parameters, such as REVAC (Nannen and Eiben, 2007). While we acknowledge that for some algorithms it is relevant to optimise along several objectives, we focus solely on single-objective configuration for the running time of NP-hard problems. This restriction, which corresponds to many practical applications of algorithm configuration, allows us to use well-studied configuration scenarios from the library AClib (Hutter et al., 2014a). Moreover, the analysis of our results is easier to interpret, without the added complexity of a Pareto front analysis. For many such scenarios, we have no access to anytime performance or any way to approximate

a result with a lower budget, which excludes configurators relying on this kind of information, such as BOHB (Falkner et al., 2018) and Optuna (Akiba et al., 2019). We also limited our use of proprietary configurators that are not readily available for use, such as GGA (Ansótegui et al., 2009).

For our performance comparisons (see Chapter 3), we thus consider the following configurators: paramILS (Hutter et al., 2009), GGA++ (Ansótegui et al., 2015), irace (Birattari et al., 2010), SMAC (Hutter et al., 2011b) and GPS (Pushak and Hoos, 2020). While there might be more available, we believe that this selection covers prominent fundamental methods (evolutionary computation, local search, Bayesian optimisation and racing). In the following, we delve deeper into the inner workings of these configurators.

2.2.2 Details on prominent AAC procedures

As mentioned earlier, the AAC problem presents numerous challenges, and each configurator approaches them in a different way. To handle the extensive running time of the target algorithm, configurators can use an empirical performance model (*e.g.* SMAC and GGA++) that allows them to estimate the running time based on features characterising the instances, similarly to the work done in algorithm selection (AS) (Xu et al., 2008; Lindauer et al., 2015). Capping mechanisms reduce the number of algorithm runs by stopping the evaluation of poorly performing configurations (*e.g.* paramILS and irace). To handle the vast search space, numerous search algorithms are available in the literature. Each configurator is based on well-known search strategies that have already demonstrated success in other fields, such as genetic algorithms, estimation of distribution, or golden search. Each configurator considered in our work is described in more detail in the following.

paramILS

paramILS (Hutter et al., 2007, 2009) is the earliest method that we consider. It is an iterated local search (ILS) algorithm (Lourenço et al., 2003), designed to optimise the parameters of heuristic algorithms. Following ILS, it starts from a point in the search space and, through small perturbations, explores its neighbourhood. It restarts this process several times to collect a set of good configurations and returns the best one when the configuration budget is exhausted. Additionally, it extends ILS with an adaptive capping mechanism that limits the time spent on less promising sets of parameters.

The high-level procedure of paramILS is described in Algorithm 2.1. ParamILS is initialised with random configurations (line 1) from which it keeps the one determined to perform best according to $\mathcal{M}$. It perturbs this configuration with small changes to one parameter at a time and retains the first found improvement (line 2). Then, until the assigned budget is exhausted, it applies s perturbations to the configuration (line 5) and improves it by taking the first neighbouring configuration that improves upon it (line 6). If the best found configuration so far, ω_{inc}, is outperformed by the newly found configuration, its value is updated (line 7). paramILS restarts the local search at random with a probability of $p_{restart}$ set to 1% by default (line 5).

Hutter *et al.* propose two approaches to decide, line 7, which of the two configurations performs best. This leads to two variants of the paramILS procedure. The first variant, basicILS, compares directly the performance of two configurations on a fixed subset of instances. The second variant, focusedILS, gradually increases the number of runs performed with the configuration that was evaluated the fewest times, until one configuration dominates the other (ω_1 dominates ω_2 if it performs at least as well on all instances evaluated with ω_2). At each iteration of the core loop (lines 3 to 9), focusedILS increases the number of instances on which ω_{inc} is evaluated. Moreover, to avoid spending excessive time evaluating poorly performing configurations, an adaptive capping mechanism is employed. If, during a target algorithm run, the perturbed configuration is deemed to perform significantly worse than the current best, the run

Algorithm 2.1 ParamILS

Input Ω: configuration space, r: number of initial configurations, $p_{restart}$: probability of restarting, s: number of perturbations, *Budget*: the maximum time allowed to find a configuration.
Output ω_{inc}: the best found configuration.

1: Sample r random configurations in Ω, set ω to the best-performing.
2: Perform iterative first improvement on neighbourhood of ω (the configurations that differ by one parameter) and set ω_{inc} to the result
3: **while** *Budget* not exhausted **do**
4: $\omega \leftarrow \omega_{\text{inc}}$
5: Apply s random perturbations (change one parameter value) to ω
6: Perform iterative first improvement on neighbourhood of ω (the configurations that differ by one parameter)
7: If ω is better that the best known configuration ω_{inc}, update ω_{inc}
8: With probability $p_{restart}$, reset ω to a random configuration
9: **end while**
10: **Return** ω_{inc}

is terminated. According to the evaluation presented by (Hutter et al., 2009), this reduced up to 10-folds the required configuration time.

GGA++

GGA (Ansótegui et al., 2009) is a genetic algorithm that considers a population in which each individual is a configuration and the performance of the target algorithm is used as the fitness function. Following the work of Lis and Eiben (1997), the individuals are assigned a "gender" that separates them into two groups: the competitive and the non-competitive. Among the competitive group, only the $c\%$ best individuals can produce offsprings, based on a tournament evaluating their fitness on a subset of instances from the training set. Among the non-competitive group, a portion of randomly selected individuals will produce offsprings (their number is tailored to keep the size of the population stable). This approach allows to minimise the number of direct evaluations of configurations and to keep diversity in the population thanks to the non-competitive individuals. Following standard practice in the field, Ansótegui *et al.* apply crossovers and mutations to the offspring produced at each generation, and kill individuals when they reach the age limit (set to 3 by default).

Later, Ansótegui et al. (2015) proposed GGA++. They added a random forest surrogate model to predict the performance of the non-competitive individuals and

Algorithm 2.2 GGA++

Input Ω: configuration space, *Age*: maximum age of the population, c: portion of the individuals kept through competition, g: portion of the offspring genetically engineered, r: portion of individuals replaced by random at each generation, *Budget*: the maximum number of generations allowed to find a configuration.
Output ω_{inc}: the best found configuration.

1: pop $\leftarrow$ SampleUniform(Ω) ▷ Initialise the population.
2: Assign each individual an age (from 1 to *Age*)
3: separate into $\text{pop}_{competition}$ and $\text{pop}_{attraction}$
4: **while** *Budget* not exhausted **do**
5: $\quad \text{pop}_{\text{best}} \leftarrow$ best $c\%$ of $\text{pop}_{competition}$
6: $\quad \text{pop}_{\text{new}} \leftarrow$ offsprings of pop_{best} and $\text{pop}_{attraction}$ with probability g of genetic engineering
7: $\quad$ Mutate offspring.
8: $\quad$ Replace portion r of the non-competitive population with random individuals
9: $\quad$ Evaluate pop_{best} and update the incumbent ω_{inc}
10: **end while**
11: **Return** ω_{inc}

used it as a measure of attractiveness to select which of them would mate. This surrogate model also provides a heuristic to predict the performance of possible offsprings and decide which one should be produced. The model is specifically tuned to be more precise in the top-performing areas of the search space, and is used at several steps of the algorithm (Algorithm 2.2). Compared to GGA, competitive individuals are still selected based on a tournament (line 4), but the non-competitive individuals are selected with a probability based on their predicted performance instead of uniformly at random. Moreover, a portion g (set to 1 in the original publication) of the offsprings are built using genetic engineering, meaning that the surrogate model indicates which offspring should be produced by the mating process (line 5). This engineering process, coupled to the selection of parents based on the same surrogate model, would lower the diversity of the population, which is key in evolutionary computing for balancing exploration and exploitation. Thus, part of the population is replaced by random individuals at each generation (line 9).

irace

The configurator irace (Birattari et al., 2010) is based on racing methods, drawing from prior work by Maron and Moore (1997). The idea, as seen in Algorithm 2.3, is to perform a race among a set of configurations, *i.e.*, to run them incrementally on more and more instances, dropping configurations as soon as they have been found to perform statistically worse than others. Then, irace generates new configurations using an estimation of distribution mechanism (see *e.g.* Hauschild and Pelikan, 2011) that

Algorithm 2.3 irace (Based on Algorithm 1 from Birattari et al. (2010))

Input $I = [I_1, I_2, \ldots] \in \mathcal{I}$, Ω: the parameter space, $\mathcal{M}(\omega, I) \in \mathbb{R}$: performance measure, $Budget$: the maximum number of iterations allowed to find a configuration.
Output Ω_{elite}: a set of best found configurations.

1: $\Omega_1 \leftarrow$ configurations sampled uniformly at random from Ω
2: $\Omega_{\text{elite}} \leftarrow$ elites of the race among Ω_1
3: $j \leftarrow 1$
4: **while** $j < Budget$ **do**
5: $\quad j \leftarrow j + 1$
6: $\quad \Omega_{\text{new}} \leftarrow$ configurations sampled from Ω around Ω_{elite}
7: $\quad \Omega_j \leftarrow \Omega_{\text{new}} \cup \Omega_{\text{elite}}$
8: $\quad \Omega_{\text{elite}} \leftarrow$ elites of the race among Ω_j
9: **end while**
10: **Return** Ω_{elite}

builds a probabilistic model to capture the distribution of promising configurations based on the best configurations seen in the previous race, relying on the assumption that good configurations are likely near each other within the configuration space.

The races (lines 2 and 8) proceed as follows: each configuration is evaluated on a fixed number of instances T^{first}; then, after each T^{each} evaluations on a new instance, a Friedman two-way analysis of variance by ranks (with a significance level of 0.05 by default) is applied to discard less performing configurations. By default, $T^{first} = 5$ and $T^{each} = 1$. The irace procedure does not perform multiple hypothesis testing correction, which leads to a higher probability to mistakenly discard configurations.

To sample new configurations (line 6), parent configurations are first sampled with a higher probability for higher-ranked configurations. Then, children configurations are sampled nearby following a truncated normal distribution centred around their parent configuration and with a standard deviation σ_d^j for parameter d at iteration j. To sample configurations increasingly closer to known elites, at each iteration this standard deviation is updated according to $\sigma_d^j = \sigma_d^{j-1} \cdot \left(\frac{1}{N_j^{new}}\right)^{1/N^{param}}$, where N_j^{new} is the number of new configurations sampled in iteration j and N^{param} is the number of parameters.

Later, inspired by paramILS previously described, it was extended with a capping mechanism (López-Ibáñez et al., 2016; Cáceres et al., 2017) to stop evaluating configurations as soon as they are clearly worse performing than the elites.

SMAC

SMAC (Hutter et al., 2011b; Lindauer et al., 2022) is based on a sequential model-based optimisation approach (also known as Bayesian optimisation). Bayesian optimisation allows for the replacement of costly evaluation of the real performance with a surrogate model that estimates the performance of configurations cheaply (see *e.g.* Mockus, 1989). As shown in Algorithm 2.4, SMAC constructs a random forest surrogate model to predict the performance of configurations on given instances (line 5). It samples random configurations (lines 6 and 8), and some of them are optimised using local search on the expected improvement obtained from the predictions of the model (line 7). The local search procedure uses a notion of expected improvement to produce promising configurations. This expected improvement is defined as the expected difference between the performance of the best known ω_{inc} and the performance of the sampled configuration. To ensure diversity in the evaluated configurations, this local search procedure is applied only to half of the configurations. The configurations found

through local search are then sorted from highest to lowest expected improvement, and interleaved with those random configurations (line 9). During the intensification phase (line 10), the challenger configurations thus found are compared to the incumbent, first by running them on the same instances as those on which ω_{inc} has already been run, and then by adding more instances randomly to slowly increase the set of instances on which the runs are collected and avoid overfitting to a subset of instances from the training set. To avoid spending too much time on non-promising configurations, as soon as the challenger configuration ω_{ch} has used up as much time budget as ω_{inc} needs for all instances it has been evaluated on, the evaluation of ω_{ch} is stopped.

GPS

More recently, GPS (Pushak and Hoos, 2020) introduced a new, highly parallelisable search approach, described in Algorithm 2.5 (we omit the steps related to the queues and workers management and focus on the configuration aspect of the procedure). GPS relies on two fundamental assumptions. The first assumption is that the interactions between parameters are limited. This allows each parameter to be optimised independently and the incumbent configuration ω_{inc} to be update parameter per parameter (line 10). The second assumption, as per the findings of Pushak and Hoos (2018), is that the response landscape of the solvers is unimodal. This assumption led

Algorithm 2.4 SMAC

Input Ω: configuration space, ω_d: the default configuration, *Budget*: the maximum time allowed to find a configuration.
Output ω_{inc}: the best found configuration.

1: $\mathcal{R}$: target algorithm runs performed, M: performance model.
2: $\omega_{\text{inc}} \leftarrow \omega_d$
3: $\mathcal{R} \leftarrow run\,(\omega_d)$
4: **while** *Budget* not exhausted **do**
5: $M \leftarrow$ update model M from $\mathcal{R}$
6: $\Omega_{\text{prom}} \leftarrow$ sample configurations from Ω uniformly
7: $\Omega_{\text{prom}} \leftarrow$ refine Ω_{prom} using local search on the expected improvement derived from M
8: $\Omega_{\text{rand}} \leftarrow$ sample configurations from Ω uniformly
9: $\Omega_{\text{new}} \leftarrow$ interleave Ω_{rand} and Ω_{prom}
10: $\omega_{\text{inc}} \leftarrow$ compare configurations from Ω_{new} to ω_{inc} until a better one is found or a time limit is reached
11: **end while**
12: **Return** ω_{inc}

to the use of the golden search algorithm from by Kiefer (1953), which uses the golden ratio to select interior points within an interval and discards subintervals where the extremum cannot lie.

For each parameter p, GPS maintains a bracket of possible values instead of keeping the entire domain $\mathcal{D}_p$. It uses the golden search algorithm to adjust this bracket, such that it is expected to still contain the optimum value (line 13). Since golden search only works for strictly unimodal functions, a property that is likely to be violated for some AAC scenarios, GPS includes a mechanism to expand back the bracket thus reduced. To accept new values for the parameters, it uses a permutation test with significance value $\alpha = 0.05$ (lines $9 - 10$). It evaluates configurations on parallel cores, prioritising the most promising parameters using a bandit approach (line 8). It gradually increases the number of instances on which configurations are evaluated (lines 14 and 16), while using an adaptive capping mechanism to prevent excessive time spent on poorly performing configurations. GPS is the most recent algorithm included in our comparison. It combines mechanisms from earlier works, including adaptive capping from paramILS, racing from irace and the intensification approach from SMAC.

Algorithm 2.5 GPS

Require: numInitInst: Initial number of instances, instIncr: Instance increment.

1: Initialise incumbent ω_{inc} with default
2: **for** each parameter p **do**
3: Initialise bracket B_p
4: Initialise $\mathcal{I}_p$ with numInitInst random instances
5: Queue a run for default value $\omega_{\text{inc}}[p]$
6: **end for**
7: **while** *Budget* not exhausted **do**
8: Sample parameter p using bandit queue
9: **if** $\exists v$ such that $m(\omega_{\text{inc}}|_{p=v}) \prec_\alpha m(\omega_{\text{inc}})$ **then** ▷ based on a permutation test
10: $\omega_{\text{inc}}[p] \leftarrow v$ ▷ Update incumbent
11: **end if**
12: **if** sufficient evidence for improvement **then**
13: Adjust bracket B_p
14: Add instIncr random instances to $\mathcal{I}_p$
15: **else if** each $v \in B_p$ has been run on each $I \in \mathcal{I}_p$ **then**
16: Add instIncr random instances to $\mathcal{I}_p$
17: **end if**
18: Queue new target algorithm runs for B_p
19: **end while**
20: **return** ω_{inc}

Table 2.1: List of scenario characteristics

Category	Name	Domain	Description
Algorithm	Deterministic	Boolean	Is the algorithm deterministic or does it need to run with several seeds
Parameters	Number	$\mathbb{N}+$	Number of parameters
Parameters	Categoricals	$\mathbb{N}+$	Number of categoricals parameters
Parameters	Integers	$\mathbb{N}+$	Number of integers parameters
Parameters	Continuous	$\mathbb{N}+$	Number of continuous parameters
Parameters	Conditionals	$\mathbb{N}+$	Number of Conditionals parameters
Parameters	Defaults	$\mathbb{N}+$	Number of default configurations
Parameters	Forbidden	$\mathbb{N}+$	Number of forbidden configurations
Instances	Origin	String	Origin (generated, real-world)
Instances	Training size	$\mathbb{N}+$	Number of training instances
Instances	Testing size	$\mathbb{N}+$	Number of testing instances
Instances	Features	$\mathbb{N}+$	Number of features per instance
Instances	Clusters	$\mathbb{N}+$	As a measure of dataset homogeneity
Scenario	Cutoff time	$\mathbb{N}+$	Maximum running time before interrupting the algorithm run
Scenario	Timeouts	$[0, 100]$	Percent of training instances that timeout with the default configuration
Scenario	Budget	$\mathbb{N}+$	Time given to the configurator

2.3 Configuration scenarios

In this section, our goal will be to answer RQ1 – *How can the configuration scenarios be best described and characterised?*. To do so, we list scenarios from the literature and define characteristics to compare them, as well as the performance of configurators on these scenarios. Based on the definition introduced in Section 2.1.1, a configuration scenario would need to describe the target algorithm A with possibly a cutoff time T_{cut}, a configuration space Ω with possibly a default configuration, a set of problem instances $\mathcal{I}$, and a metric to optimise $\mathcal{M}$. The considered list of characteristics is listed in Table 2.1. We attempt to be representative of the problems considered in this thesis, but do not claim to be exhaustive.

2.3.1 Benchmark instance sets

The configuration scenarios we considered throughout this thesis are based on ten sets of randomly generated and real-world instances across four NP-hard problems:

Table 2.2: Benchmark instance sets characteristics.

	Name	Origin	*Training size*	*Testing size*	*Features*	*Clusters*
SAT	CF	generated	298	301	113	14
	LABS	generated	350	350	119	6
	UNSAT	generated	299	249	113	12
Planning	Satellite	generated	2000	2000	305	7
	Zenotravel	generated	2000	2000	305	5
MIP	CLS	generated	50	50	148	3
	COR-LAT	real-world	1000	1000	148	19
	RCW2	real-world	495	495	148	6
	REG200	generated	999	999	148	2
TSP	rue-1000-3000	generated	50	250	64	9

Boolean satisfiability (SAT), automated planning (AI planning), mixed integer programming (MIP) and traveling salesperson problem (TSP). They are all part of the library AClib introduced by Hutter et al. (2014a) and have been widely used in the papers introducing new AAC methods and related mechanisms. We used the training and testing sets as provided by AClib. A more detailed description of each of those datasets is provided below, and their characteristics are listed in Table 2.2. The number of clusters is computed based on the instance features using the mean-shift (Comaniciu and Meer, 2002) implementation of `scikitlearn`. Each cluster is thus a group of instances which are close to each other in the feature space according to their Euclidean distance.

Boolean satisfiability

The SAT problem is a classical mathematical problem that has been proven to be NP-complete in the 1970s (Cook, 1971). Given a Boolean formula, the goal is to either find an assignment of truth values to its variables that satisfies the formula or prove that it is unsatisfiable. Such Boolean formulas can describe many real-world problems, and thus the SAT problem appears in a wide range of applications – such as model checking (see *e.g.* Biere et al., 1999), software and hardware verification (see *e.g.* Burch et al., 1994) or automated theorem proving (see *e.g.* Brown, 2013).

We use three SAT benchmarks that originate from the configurable SAT solver challenge (Hutter et al., 2017) and have been widely used in the subsequent AAC literature (see *e.g.* Lindauer and Hutter, 2018; Pushak and Hoos, 2020): a set of instances generated by a CNF fuzzing tool (CF) (Brummayer et al., 2010), a set of low auto-correlation binary sequence problems converted into SAT (LABS) (Mugrauer and Balin, 2013) and a set of 5-SAT problems generated uniformly at random from which only unsatisfiable instances have been kept (UNSAT).

Automated planning

AI planning aims at generating a sequence of actions to reach a goal. It has many applications to real-world problems, such as cyber security (see *e.g.* Boddy et al., 2005) or maintenance scheduling (*e.g.* Verbert et al., 2017). While AI planning scenarios have rarely been used in work introducing new AAC procedures, they were in analysis papers such as the work of Fawcett and Hoos (2016) and the prominence of AI planning led us to include those benchmarks

Our two automated planning benchmarks originate from the third International Planning Competition (Long and Fox, 2003) and were included in the analysis of Fawcett and Hoos (2016); a set about the control and observation scheduling of satellites (Satellite) and a set of route planning problems (Zenotravel) (Penberthy and Weld, 1994).

Mixed integer programming

The mixed integer programming problem is a well-studied constraint programming problem, in which we try to minimise or maximise an objective function while satisfying constraints on a given set of variables. Many problems can be formulated as instances of MIP, which leads to a wide range of applications.

Three of our four MIP benchmarks originate from a study on MIP solver configuration (Hutter et al., 2010a) and were included in follow-up work (see *e.g.* Ansótegui et al., 2015; Pushak and Hoos, 2020): a set of capacitated lot-sizing benchmark (CLS) (Atamtürk and Muñoz, 2004), a set of MIP problem instances generated with the combinatorial auction test suite (Leyton-Brown et al., 2000) and a set of real-life data for wildlife corridors for grizzly bears in the Northern Rockies (Gomes et al., 2008). The fourth benchmark stems from work on combining AAC and AS (Xu et al., 2011): a set of MIP-encoded habitat preservation data for the endangered red-cockaded woodpecker (Ahmadizadeh et al., 2010).

Table 2.3: Characteristics of the target algorithms.

Name	*Deterministic*	*Total*	*Categoricals*	*Integers*	*Continuous*	*Conditionals*	*Defaults*	*Forbidden*
Clasp	False	70	33	30	7	60	1	2
Lingeling	False	322	137	185	0	0	1	0
SpToRiss	False	222	170	36	16	190	1	21
Kissat	False	133	61	72	0	0	1	0
LPG	False	67	48	5	14	25	1	12
CPLEX	True	73	50	16	7	4	1	0
LKH	False	23	11	12	0	3	1	0
EAX	False	2	0	2	0	0	1	0

Traveling salesperson problem

The TSP is a well-studied optimisation problem. Given a set of points and the distance between them, one aims to find the shortest tour that visit each of the points at most once and returns to the starting point. Other than its application to package delivery, it is also widely used in industry for circuit engraving, for example (Matai et al., 2010).

Our benchmark contains two sets of generated random uniform Euclidean instances originally introduced with irace Ansótegui et al. (2009) and subsequently used in the GPS paper (Pushak and Hoos, 2020).

2.3.2 Target algorithms

In our experiments, we used prominent solvers for SAT, AI planning, MIP and TSP (see Table 2.3). In the following, we explain those choices.

Boolean satisfiability

Our SAT solvers were selected based on their performance in the Configurable SAT Solver Challenge (CSSC) 2014 (Hutter et al., 2017) : Lingeling (Biere, 2014) ranked first on the *industrial SAT+UNSAT* track and second on the *crafted SAT+UNSAT* track, Clasp (Gebser et al., 2012) first on the *crafted SAT+UNSAT* and *Random SAT+UNSAT* tracks and SparrowToRiss (SpToRiss) (Balint and Manthey, 2014) second on the *Random SAT* track. Because there have been advances in SAT solving

since then, we also added Kissat (Balyo et al., 2020), which was the winner of the SAT 2020 competition, is highly configurable, and is similar to CadiCal (Biere et al., 2020), which is known to benefit from configuration (Pushak and Hoos, 2020).

Automated planning

For automated planning, we selected LPG (Gerevini and Serina, 2002; Gerevini et al., 2003, 2008, 2011), as it has been successfully configured previously (Vallati et al., 2013; Fawcett and Hoos, 2016) and is also available through AClib. While there has been further development in AI planning, LPG has been very impactful in the field, as shown by the ICAPS 2019 Influential Paper Award received by Gerevini and Serina (2002).

Mixed integer programming

For MIP, we chose IBM's CPLEX solver, as it is widely used in practice and has shown great potential for performance improvement through AAC (Hutter et al., 2010a). We use version 12.6, since it is the version used by Hutter et al. (2010a) and thus included in AClib. CPLEX is widely used in the literature since it provides a state-of-the-art MIP solver free of charge for academic use.

Traveling salesperson problem

For TSP, we use LKH (Helsgaun, 2000) – an implementation of the Lin-Kernigan Heuristic – and EAX (Nagata and Kobayashi, 2013) – a Genetic Algorithm –, two prominent TSP solvers that have been configured in the study introducing GPS (Pushak and Hoos, 2020). They are considered to be state-of-the-art heuristic solvers (see *e.g.* Heins et al., 2024).

2.3.3 Scenarios

For all scenarios from AClib, we followed the setup defined there, including default configurations. As a configuration objective, we used minimisation of PAR10 (the average running time of the target algorithm, with timed-out runs counted as ten times the cutoff time). Moreover, for the MIP scenarios, we duplicated the scenarios to run them with a lower cutoff time, following the study introducing irace (Cáceres et al., 2017), in order to evaluate more closely the impact of the cutoff time on the performance of the configurators. The details are described in Table 2.4.

Table 2.4: Characteristics of the scenarios.

Benchmark	Algorithm	Cutoff [s]	Timeouts [*count (%)*]	Budget [s]
CF	Clasp	300	8 (2.7)	172800
	Lingeling		12 (4.0)	
	SpToRiss		16 (5.4)	
	Kissat		8 (2.7)	
LABS	Clasp		81 (23)	172800
	Lingeling		85 (24)	
	SpToRiss		89 (25)	
	Kissat		79 (23)	
UNSAT	Clasp		0 (0.0)	172800
	Lingeling		0 (0.0)	
	SpToRiss		0 (0.0)	
	Kissat		0 (0.0)	
Satellite	LPG	300	0 (0.0)	172800
Zenotravel			0 (0.0)	
CLS	CPLEX	10000	0 (0.0)	172800
COR-LAT			1 (0.1)	
RCW2			0 (0.0)	
REG200			0 (0.0)	
CLS	CPLEX	300	0 (0.0)	172800
COR-LAT			3 (0.3)	
RCW2			15 (3.0)	
REG200			0 (0.0)	
rue-1000-3000	LKH	86	12 (24)	86400
	EAX		7 (14)	

Table 2.5: Statistics on scenario characteristics

Category	Name	Domain	Min	Max	Median
Parameters	Number	$\mathbb{N}+$	2	322	71.5
Parameters	Categoricals	$\mathbb{N}+$	0	170	49
Parameters	Integers	$\mathbb{N}+$	2	185	23
Parameters	Continuous	$\mathbb{N}+$	0	16	3.5
Parameters	Conditionals	$\mathbb{N}+$	0	190	3.5
Parameters	Defaults	$\mathbb{N}+$	1	1	1
Parameters	Forbidden	$\mathbb{N}+$	0	21	0
Instances	Training size	$\mathbb{N}+$	50	2000	422
Instances	Testing size	$\mathbb{N}+$	50	2000	422
Instances	Features	$\mathbb{N}+$	64	305	148
Instances	Clusters	$\mathbb{N}+$	2	21	6.5
Scenario	Cutoff time	$\mathbb{N}+$	86	10000	300
Scenario	Timeouts	$[0, 100]$	0	1.5	25
Scenario	Budget	$\mathbb{N}+$	86400	172800	172800

Analysis

An overview of the salient characteristics of the AAC scenarios is given in Table 2.5.

Our set of scenarios shows a wide variety of search spaces with various numbers of parameters in each category. We note that, while some configurators can handle several default configurations as a starting point, all of our scenarios have only one default configuration, which is the case for most solvers, but could be extended by using configurations found to be well-performing in the literature. Additionally, the scenarios we consider contain only one deterministic algorithm. While this is a small number, it is also representative of NP-hard solvers, which often rely on stochasticity to more effectively search for solutions.

The number of instances in the datasets varies. We note that all scenarios come with instance features, which are exploited by some but not all configurators. This give an advantage to configurators that are using those features, *e.g.* to learn a surrogate model. We note that most of our datasets are artificially generated, albeit in such a way that they capture important characteristics of real-world applications. Also, almost all our benchmarks have an equal number of instances in the training and testing sets. This allows the testing set to contain a large variety of instances and have a testing performance that would better represent the distribution of instances, while having enough variety in the training set to avoid overfitting.

Finally, the scenarios show little diversity in their cutoff times and overall wall-clock time budget. However, this is a feature of AClib, and our decision to follow AClib's approach limits us in this aspect.

2.4 Conclusion

In this chapter, we introduced the Automated algorithm configuration problem. We highlighted three main challenges of this problem: the expensive evaluations of the target algorithm, the complexity of the configuration space, and the variability. Then, we described standard methods for solving it. We described a large and varied set of scenarios and datasets to be used in the remainder of this thesis and characterised them by means of a set of features, in an attempt to highlight the ways in which they relate to or differ from each other. The following chapter will provide an in-depth evaluation of the AAC methods introduced here.