Evolution of Algorithm Portfolio for Solving Strategies

Alice Tarzariol

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine, Italy

Abstract. The last decades have witnessed tremendous growth in the performance of solvers dealing with *NP-hard* problem instances. Commonly, powerful solvers outperform the other candidates for the majority of cases but obtain poor results for particular sub-domains. For this reason, *Algorithm Portfolio* approaches have been developed, allowing to exploit a set of solvers with complementary strengths, instead of running a single best one for all instances. Recent successful implementations combine these approaches with automatic *Algorithm Configuration* methods, which consider a single parameterized solver and identify the best putative values of its hyper-parameters for the analyzed instances distribution. This paper aims to illustrate the development of the principal approaches that leverage machine learning to accelerate the solution search of modern solvers. Furthermore, it presents the basic notions necessary to understand the process that enabled these techniques to achieve their amazing results with some of the many performing implementations for *SAT, Answer Set Programming*, and *Constraint Programming*.

Keywords: Algorithm Portfolio, Algorithm Configuration, Machine Learning, Solving Strategies.

1 Introduction

Many "difficult" problems that we encounter nowadays, especially in the logistic area of industrial domains, can be modeled and solved using powerful ad-hoc tools. For example, we can reduce the considered problems in *SAT* formulae [39], programs expressed in *Answer Set Programming* [12] or *Constraint Programming* [43], and then leaving the search of solution to the implemented solvers.

Till the 2000s, the principal approach to solve a problem instance contained in a collection of heterogeneous elements consisted of running a single "best" solver that outperforms the other candidates in the majority of cases. In [26], this is called *winner-takes-all* approach, to emphasize that the algorithm that obtains the higher average performance on the whole representative instances set, will always be preferred to solve a new problem drawn from the considered distribution. Since usually the performance of algorithms highly vary from instance to instance [48], even on the same domain, systems based on *Algorithm Portfolio* methods have been subsequently developed to leverage a group of solvers instead of a unique resource. They obtained outstanding results in industrial and solving competitions, exploiting solvers with high running time for the majority of the instances but with excellent performance on particular problem sub-domains.

Modern solvers rely on different techniques, ranging from incomplete search methods, e.g., local search, to complete ones, as for instance CDCL [35], which has been particularly useful to accelerate *SAT* and *ASP* searching engine [10, 34]. Besides the solving core, the hyper-parameter values further guide the behavior of algorithms, whose tuning can be manually performed by domain experts involving considerable time and human efforts. The period on which algorithm portfolio appeared on solving competitions coincides with the concept of *Algorithm Configuration*, i.e., the techniques that efficiently explore solvers' hyper-parameter configurations, detecting automatically the best setting for the considered domain.

This work presents a picture of the evolution of some of the principal *Algorithm Portfolio* and *Configuration* techniques implemented in nowadays solvers, providing the essential machine learning and statistical concepts necessary to address this topic. Since many different implementations are available in the literature, we solely report a limited portion of meaningful examples, useful to understand the main concepts and practical issues addressed by these systems. For a more comprehensive review we refer to [4, 24, 25, 46].

The structure of this paper is the following: Section 2 provides the basic machine learning concepts and related considerations for this topic, then Section 3 contains the definition and characteristics of *Algorithm Portfolio* methods, together with some practical examples derived from different fields, like SAT, ASP and CP. These are just some of the main programming paradigms implemented since in the literature we can find examples applied in further contexts, e.g., Planning, Mixed Integer Programming, etc. In Section 4, we will see the principles and main examples of *Algorithm Configuration* and lastly, Section 5 contains several modern systems that combine the previous ideas to speed up the search.

2 Machine Learning Background

Although Machine Learning presents myriad of application domains, this section defines solely the techniques and notions that concerns our topic.

Several criteria can be considered to classify the learning techniques [45], such as the prior knowledge available, determining *deductive* and *inductive learning*. The former deduces a logical conclusion from a set of "certain" general rules: CDCL implements this approach since, given an input formula as prior knowledge, it leads to an equisatisfiable formula by adding clauses that possibly accelerate the search. The learned clauses are certain, however, the scope of the deduced components is limited to solving the single problem instance considered. On the other hand, the prior knowledge of the second approach provides a support to draw conclusions, therefore, the truth of the inferred components follows a certain probability. These methods are leveraged by algorithm portfolio and automatic configuration systems, using as prior knowledge a set of previously solved problem instances, possibly independent and identically distributed according to some fixed probability distribution. An additional stage called *features extraction*, pre-processes the original instances by mapping them into a space defined by a set of *features*, i.e., measurable properties or characteristics. The features of the considered works include *static features* derived from syntactic analysis, e.g., the number of clauses in a SAT instance and statistics like clauses and variables ratio, or they can also involve *probing features*. The last features are more informative but also more expensive to be calculated since they are obtained by probing parts of the search space for a brief time interval. Finally, *dynamic features* are directly computed while solving is taking place: we do not contemplate this kind of features, although several portfolio methods rely on them, as for instance [7].

A further classification criterion regards the component learned; for instance, we could aim to find a direct mapping f from an input domain $\mathcal{I}$ into an output space $\mathcal{O}$. In case of *regression*, f should describe the relationships between an independent *continuous* variable $o \in \mathcal{O}$ and a set of dependent variables $i \in \mathcal{I}$, e.g., a function that predicts the performance of an algorithm for a specific problem instance, given its features. Instead, if the output domain $\mathcal{O}$ is discrete, the task is called *classification*, e.g., the predicted value could directly return the most suitable solver to run for the considered instance.

As concerns *when* processing the prior knowledge, the *eager learning* approach builds a general model from the data *offline*, i.e., before observing a new instance. Alternatively, the *lazy learning* method processes the representative input instances *online*, namely when the system encounters a new element to solve [36]. Moreover, considering the type of feedback provided to learn a model that entails f, we can distinguish between *supervised learning* and *unsupervised learning* methods. The former learn a model from a set of instances whose label (i.e., the correct output) is specified, trying to minimize the value produced by the selected *loss function*. In contrast, the latter finds a series of patterns in the input collection, since their output values are missing. Lastly, a third approach called *reinforcement learning*, iteratively learns a series of actions to take in a specific environment, in order to maximize a cumulative reward, trading off between exploration and exploitation phases. Although the literature includes different approaches, this paper analyzes just supervised-learning systems.

Learning is performed by means of a *parametric model*, namely an explicit model with a fixed set of parameters that range over a defined domain. Or we can use *nonparametric model* approaches, whose structure cannot be characterized by a bounded set of parameters as its complexity grows in relation to the number of input data. The *instance-based learning* is a *nonparametric lazy learning* method since its knowledge consists just of indexed input instances and the reasoning phase over them is delayed until a new example must be classified.

Some examples of supervised learning techniques applied for regression are *linear regression, regression forests* and *support vector regression*, while for classification, we can rely for instance on *logistic regression, support vector machine* and *decision forests. Case-based reasoning* and *k-nearest neighbors* are examples of instance-based learning that can be employed for both tasks.

2.1 Model Evaluation and Selection

In Machine Learning, the *No Free Lunch Theorems* defined in [47] led to a famous implication on the choice of the learning algorithm, as they proved that for any two algorithms, determining the most accurate learner for every domain is unfeasible. Consequently, in the real application scenario, the usual approach consists of considering manifold learning algorithms, evaluate them and then select the one that leads to the best result on a separate instances collection [41].

In order to evaluate the performance of a learning system, the data set used for training, called *training set* $\mathcal{N}_{train}$, should contain instances drawn from the same distribution of the evaluation collection, i.e., the *test set* $\mathcal{N}_{test}$. However, to obtain unbiased results, the two sets should have different instances. The learned component should trade-off between correctly predicting the output value for instances in both $\mathcal{N}_{train}$ and $\mathcal{N}_{test}$, reducing as much as possible both *over* and *under-fitting* phenomena.

Once an *evaluation measure* is chosen, *model evaluation* is performed: the most common is the *holdout method* which partitions the original collection in two sets, according to a certain splits ratio. Alternatively, *k-folds cross-validation* is a more costly method that leverages the whole collection for training, as it partitions the original data into k equal sized subset and performs training and testing k times, using each time the i-th test set and then averaging the results.

Besides the parameters learned during training, the learning algorithm can have further *hyper parameters* whose values are pre-defined. Each configuration leads to a different learning model, therefore a ranking method called *model selection* orders the resulting models with respect to their relative estimated performances, selecting the one that returns the highest. For instance, the *holdout method* can be generalized to split the collection into three sets: training several learning algorithms on $\mathcal{N}_{train}$, selecting among the learned models the best-performing according the *validation set* $\mathcal{N}_{val}$ and finally evaluating it on $\mathcal{N}_{test}$. An extension of the *k-folds cross-validation* method is a valid alternative to conduct model selection as it leads to good predictions for both model and hyper parameters, despite using a relatively small data collection.

A final remark concerns the features extraction phase: the choice of significant but cheap features that adequately characterize the considered instances heavily influences the model performances. Examining a large set of features is usually counterproductive since they could contain the same information or even be misleading for the prediction. Two techniques that can reduce the number and impact of weak features are *subset selection* like *stepwise regression*, or *regularization* methods like *ridge regression* [14]. The former applies statistical techniques, like *forward-stepwise regression* that, starting from an empty set of features, iteratively includes the one that brings the best statistical improvement on the prediction model, returning at the end a subset of features that should capture the best characteristics to model the input instances. The latter approach instead, performs regression introducing a *regularization term* to the loss function of the model, allowing to keep all features but assigning to them different weights.

3 Algorithm Portfolio

Algorithm Portfolios methods allow to consider simultaneously a set of algorithms exploiting their complementary strengths across an instances domain. Their first designs are provided in [17] and [13].

Two classification criterion contained in [25] distinguish these methods considering the moment and the object chosen from the portfolio. With regard to "*when to perform the choice*", a *static portfolio* approach operates the selection offline, which lessens further overhead during solving, despite precluding adjustment of the learned model in case of poor performance. On the other hand, a *dynamic portfolio* approach allows switching the initially chosen algorithm online, increasing its flexibility as well as the computational cost during solving.

Considering the "*object selected*" to solve a new instance, *algorithm scheduling* approach defines time slices and running order for each portfolio algorithm and, if a multicore architecture is available, *parallel portfolio* method extends this idea by running the schedule in parallel. Lastly, *algorithm selection* chooses the algorithm with the highest performance to solve the considered problem instance, as we can see in Figure 1. In [26] this is called *per-instance algorithm selection*, to differentiate it from the *winner-takes-all* approaches, also called *per-distribution algorithm selection*.

Algorithm scheduling is a more robust approach than algorithm selection, as the latter chooses a unique solver and thus poor results cannot be mitigated. On the other hand, a schedule is useful if numerous problem instances are solved within short time intervals by different solvers.

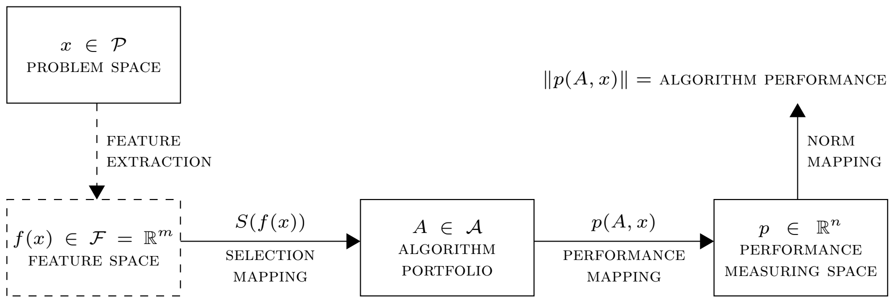

Fig. 1. Diagram of algorithm selection problem derived from [42], with additional features extraction step (dashed elements) that maps the problem instance x into its features vector $F(x) \in \mathcal{F}$ which ranges in a simpler and more restricted domain than $\mathcal{P}$.

3.1 Algorithm Selection Implementations

SATzilla One of the most famous algorithm selection implementations is the pioneer system SATzilla. From its first appearance in 2004 *SAT* competition [37], numerous successive versions have been developed [51, 52][1], improving its perfor-

[1] Just to cite the most popular.

mance so much that it won many gold medals on several solving competitions. Its success proved that algorithm portfolio implementations managed to outperform the *single best solver*, i.e., the standalone solver with the best performance across all instances, despite features extraction, selection overhead and by finding a sub-optimal solver whose performance is close to the optimal.

In [37], statistical regression techniques allowed per-instance selection by learning *offline* an *empirical hardness model* for each solver A in a defined portfolio $\mathcal{A}$. Namely, a function that predicts the amount of time that A will take to solve a problem instance i, according to its features x_i. In order to learn the *empirical hardness model* $\mathcal{M}_A$ for algorithm A using a training set $\mathcal{I}$, the following steps are performed:

1. For each instance $i \in \mathcal{I}$, compute its features together with its approximate[2] runtime for A, which is the output value that the final model should predict.
2. Next, execute in order *forward-stepwise selection* to eliminate highly correlated features, a quadratic *basis function expansion* on the new features set and then, repeat *forward-stepwise selection* on the expanded features.
3. Lastly, use *ridge regression* technique to learn the model $\mathcal{M}_A$ that predicts logarithm runtime of A for solving a new problem instance j, given x_j.

SATzilla-07 [51] extends its previous implementation, by adding further statistical techniques to the *empirical hardness model* [50] and introducing two simple but very efficient components that have been exploited in successive algorithm portfolio systems: *pre-solvers* and *backup solver*. The former is a single or a schedule of algorithms with a reasonable general performance that runs for a small amount of time before features computation. It should prevent the system performance penalization from features extraction on very easy instances, while algorithm selection focuses exclusively on harder cases. The latter consists of the solver that achieves the best average runtime on a validation data set, containing instances that are not solved by the pre-solvers. It will run if the online features computation fails for some reason (e.g., error or timeout). Lastly, SATzilla-07 applies *subset solvers selection*, choosing the best subset of solvers to use in the final portfolio by keeping the algorithms whose empirical model predicts the lowest total runtime on a validation set.

SATzilla-11 [52] follows the main structure outlined in its previous versions, but applies *decision forests* to learn classification models for algorithm selection, instead of regression. This approach uses explicit cost-sensitive loss function to weigh misclassifications in direct proportion to its impact on portfolio performance; namely, the confidence that one solver could perform well on a particular problem instance guides the choice of the algorithms (returning a discrete value since we perform classification with respect to the portfolio algorithms) while the cost-sensitive weights allow to consider how far its predicted runtime deviates from the performance of the other solvers. Thanks to its new classification approach, SATzilla-11 considerably outperforms its previous versions.

[2] As the considered SAT solvers are randomized, the running time necessaries to solve the same instance is not deterministic, but ranges in a certain interval.

Claspfolio For Answer Set Programming, CLASPFOLIO [9] follows an approach similar to SATzilla-07, performing algorithm selection among a set of twelve configurations of CLASP solver [10], with complementary strengths. It learns offline a performance model for each configuration using *support vector regression* techniques. The training set is obtained by a series of steps: initially, a collection of problem instance is grounded by GRINGO [11] and then CLASPRE, a light-weight version of CLASP, is run to extract their features. Lastly, after computing the running time needed to solve each grounded instance by every CLASP configuration, their scores are calculated and considered as labels of the training set of each configurations' model. When a new instance j is met online, CLASPRE is run to extract its features x_j but it also works as a pre-solver: if its execution leads directly to a solution, the selection is not needed. Then, the learned models use x_j to predict the scoring of each configuration and choose the best one.

ME-ASP [31] is a modular multi-engine approach for Answer Set Programming that implements algorithm selection, following a similar approach of a previous successful systems for *Quantified Boolean Formulas* (QBFs) solvers [40]. Its training set uses just static features and defines the solving time required by every portfolio solver as output values. Instead of applying an eager and parametric approach, like we have seen in the previous works, it uses *1-nearest neighbors* classification technique to retrieve the most promising solver from the portfolio to solve a new problem instance. A successive work [32] extends this approach by considering more classification techniques.

3.2 Algorithm Scheduling Implementations

The idea of using algorithm scheduling has been conceived since the heavy tailed nature of solving: often a solver either solves a problem in a short time, or it does not solve it at all within a given amount of CPU time. This behavior emerged especially on the results of several solving competitions.

The first two following solutions concern *Constraint Programming* solvers, and manage to define a schedule online, as the size of their portfolio $\mathcal{S}$ is rather small. Moreover, they rely on *instance-based learning* approach, using as knowledge base features of problem instances and the running times of $\mathcal{S}$ solvers, taken from past CSP competitions.

CPhydra [38] defines an algorithms schedule[3] assigning a different time slice to each solver in $\mathcal{S}$, according to the considered problem instance i.

After efficiently indexing its knowledge base, CPHYDRA uses the *case-based reasoning* approach to define the schedule time slices[4]. The *retrieval* phase computes the features x_i and then uses *10-nearest neighbors* techniques to

[3] Actually, the order of solvers is static since the authors aim to maximize the probability of solving new instances within the time limit.

[4] We just consider the first two core-reasoning phases of *case-based reasoning* cycle [1].

retrieve from the knowledge base, the set $\mathcal{C}$ of the ten most similar cases to i. For each $c \in \mathcal{C}$, it considers its solvers' time and similarity distance $d(c)$ with i. The *reuse* step computes the solvers scheduling, i.e., a function $f : \mathcal{S} \to \mathbb{R}$ that maps for each solver $s \in \mathcal{S}$ a time interval t that must not exceed a certain threshold T. f should be able to define a time scheduling such that, within T the number of solved instances in $\mathcal{C}$ is maximized, considering also the similarity of each case with i. We can formulate this goal as an optimization problem:

$$\begin{array}{ll} \max & \displaystyle\sum_{c \in \mathcal{C}} \frac{1}{d(c)+1} \left| \bigcup_{s \in \mathcal{S}} \quad N(s, f(s)) \right| \\ \text{subject to} & \displaystyle\sum_{s \in \mathcal{S}} f(s) \leq T \end{array}$$

where $N(s, f(s))$ contains the instances in $\mathcal{C}$ that solver s is able to solve, given at least $f(s)$ time. If every solver reaches the maximal number of solved instances within a time $\hat{t}$ such that $\sum_{s \in \mathcal{S}} \hat{t} < T$, then a schedule that allocates to each solver a time interval equal to $\hat{t}$ can trivially optimize the objective function. In this situation, in order to exploit the whole available time, CPHYDRA discards the solvers that are *dominated* by others in the portfolio and then computes the optimal time assignment for each of them, using the entire available time.

SUNNY [2] is an algorithm scheduling for *Constraint Programming* whose name derives from its characteristics: it detects a *SU*bset of portfolio solvers by using k-*N*earest *N*eighbor technique to define a laz*Y* learning model.

When a new instance i must be solved, SUNNY computes its features x_i and use them to retrieve $NN(k, i)$, i.e., the set of k most similar instances from its knowledge base. Then, it finds the minimal subset of algorithms $\mathcal{M} \subset \mathcal{S}$ that solves the highest number of instances in $NN(k, i)$ within time T (choosing the subset with the lowest average solving time in case of ties). In order to define the schedule, SUNNY divides the time window $[0, T]$ in T/σ equal-sized slots, where σ is the sum of the number of instances that each algorithm in $\mathcal{M}$ solves within T, plus those that are not solved by anyone. Then, each solver $s \in \mathcal{M}$ is assigned with a number of time slots corresponding to the number of instances in $NN(k, i)$ that s is able to solve within T. The additional time slots (that corresponds to the unsolved instances) are assigned to a previously defined backup solver. Finally the solvers are sorted in decreasing order, according to the allocated time.

A successive parallel portfolio approach [3] extends SUNNY by defining a multicore solver scheduling. It deals also with constraint optimization while, concerning CSP, it allows to obtain a dynamic scheduling that is run in parallel.

Aspeed [15] can be used for SAT, CSP or ASP solvers, and relies on CLASP [10] to determine a *"per-distribution"* schedule[5] that optimizes their performance on a set of problem instances. It was inspired by *ppfolio* [44]: a parallel portfolio

[5] As it relies on *per-distribution* approach, *online* features extraction is not required.

that won the 2011 SAT competition by simply using a handmade solver schedule based only on the number of available cores. ASPEED extends its idea by applying a more sophisticated reasoning to define a sequential or parallel scheduling.

The sequential one consists of two components: a function σ that maps a time slice to each solver in $\mathcal{S}$, and a function π that takes in input a position $p \in \{1, ..., |\mathcal{S}|\}$ and returns the p^{th} solver of the defined schedule. Considering a set of instances $\mathcal{I}$ and a $|\mathcal{S}| \times |\mathcal{I}|$ matrix R that contains the runtimes of solvers for every instances in $\mathcal{I}$, we can define both σ and π as multiobjective optimization problems. The first step consists in finding σ that optimizes the following equation:

$$\begin{array}{ll} \max & \left| \bigcup_{s \in \mathcal{S}} \{\, i \mid i \in \mathcal{I},\ R[s,i] \leq \sigma(s) \} \right| \\ \text{subject to} & \sum_{s \in \mathcal{S}} \sigma(s) \leq T. \end{array}$$

Namely, σ should allow to solve as much instances as possible in $\mathcal{I}$, considering that the time slices assigned to each solver must sum up to a time threshold T or less. As this single optimization problem leads to numerous solutions, a second objective function is considered which favors a function σ that assigns similar time slices to each solver, i.e. that optimizes

$$\min \quad \sum_{s \in \mathcal{S}} \sigma(s)^2.$$

Once the time assignment σ is defined, the solvers must be sorted in order to reduce the expected time for solving an element in $\mathcal{I}$. Thus we define a function $\pi : \{1, ..., |\mathcal{S}|\} \rightarrow \mathcal{S}$ that optimizes

$$\min \quad \sum_{i \in \mathcal{I}} \tau_{\sigma,\pi}(i)$$

where $\tau_{\sigma,\pi}(i)$ contains the time that the schedule defined by σ and π requires to solve i. The schedule follows the order defined by π: each solver s tries to solve i within its timeout $\sigma(s)$ and until i is not solved the solving times are summed up, reaching T if i remains unsolved.

The schedule can be extended to a multicore architecture by defining a function ν that, given a number of cores c, finds a partition $\mathcal{P}$ of $\mathcal{S}$ of dimension c. Then, for each solver subset $S_k \in \mathcal{P}$, it uses the previous sequential scheduling approach. The new objective function finds ν, σ and π_u for each $u \in \{1, ..., c\}$ such that the expected time for solving an element in $\mathcal{I}$ is globally minimized.

4 Automatic Algorithm Configuration

Algorithm Portfolio efficacy highly depends on the elements chosen to compose the portfolio. Besides the various techniques that underlie the solvers, there are free parameters or *hyper parameters* designed to customize their behavior by guiding further the search. Let us consider the problem of finding the optimal hyper parameters' configuration of a certain parametrized solver A with respect to a set of representative problem instances $\mathcal{I}$. Since the hyper parameters range from continuous to categorical domains[6], their combination leads to a high dimensional and structured configuration space. The exponential number of possibilities usually renders manual *parameters tuning* inefficient and without guarantee on the results quality, even if carried out by domain experts. We represent with Θ the configuration space of the hyper parameters of solver A.

Algorithm Configuration approach applies statistical techniques to automatically find the putative configuration which optimally solves problem instances that follow a certain distribution. It returns the configuration with the minimal expected cost required to solve the elements in $\mathcal{I}$ with respect to a performance metric $m : \Theta \times \mathcal{I} \to \mathbb{R}$.

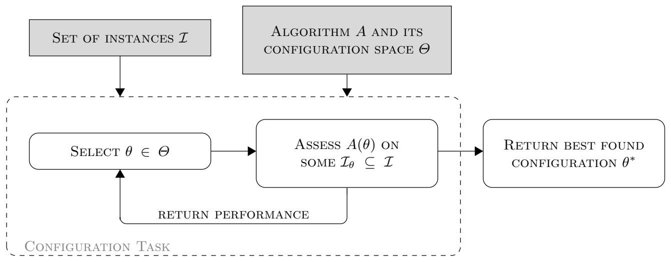
Set of instances $\mathcal{I}$
Algorithm A and its configuration space Θ
Select $\theta \in \Theta$
Assess $A(\theta)$ on some $\mathcal{I}_\theta \subseteq \mathcal{I}$
Return best found configuration θ^*
return performance
Configuration Task

Fig. 2. Diagram of Algorithm Configuration problem as contained in [27].

As we can see in Figure 2, *Algorithm Configuration* iterates two actions until detecting with sufficient confidence a configuration that overcomes the others. The former action consists in deciding which configuration chose at each step: the high dimension of Θ, even discretizing the continuous domains, renders unfeasible exploring the whole space. Therefore, efficient search strategies have been defined to entail a trade-off between diversification and intensification. ParamILS [21], for instance, uses *Iterated Local Search* to perform a biased random walk over a chain of local optima, while Gender-Based Genetic Algorithm (GGA) [5] relies on *Genetic Algorithms* to compare populations of configurations. On the other hand, Sequential Model-based Algorithm Configuration (SMAC) [19] and Iterated F-race [6] apply *Sequential Model-Based Optimization* techniques which consist in leaning a model to guide the selection of promising configurations and it exploits the gathered data to iteratively bias the model.

The latter action regards how to estimate the configurations performance and comparing them: for each configuration θ we consider an approximation of its estimated cost over a finite set of instances $\mathcal{I}_\theta \subset \mathcal{I}$ represented by $c_{\mathcal{I}_\theta}(\theta)$. We can see its definition in the objective function of *Algorithm Configuration*:

[6] Furthermore, a parameter could be *conditional* on another one, such that the former is activated just for a specific value of the latter.

$$\theta^* \in \operatorname*{arg\,min}_{\theta \in \Theta} \overbrace{\frac{1}{|\mathcal{I}_\theta|} \sum_{i \in \mathcal{I}_\theta} (m(\theta, i))}^{c_{\mathcal{I}_\theta}(\theta)} .$$

BasicILS[7] applies the easiest solution to define each $\mathcal{I}_\theta$, i.e., using the same set of N instances, $\mathcal{I}_N$, for each configuration. Unfortunately, two problems arise in the general case, depending on the chosen size: when N is too small, the cost estimated for the best putative configuration $c_{\mathcal{I}_N}(\theta^*)$ may underestimate its real cost, therefore the real optimal solution could be discarded in favor of θ^*. In contrast, if N is too high, the huge number of runs slows down the evaluation for every single configuration. Similarly, in GGA the number of instances run to evaluate each element increases linearly at each iteration.

On the other hand, FocusedILS, SMAC and Iterated F-race asses the performance over a dynamic number of evaluations, in order to quickly discard bad configurations and saving the evaluation budget for the most promising solutions by leveraging *Racing Algorithm* concept [33]. Moreover, adaptive capping techniques have been introduced to ParamILS [20] and SMAC [18] to terminate earlier runs for poor configurations.

5 Combination of Previous Methods

The following section contains some systems that combine the previous techniques: CLASPFOLIO 2 leverages both algorithm portfolio approaches, while AUTOFOLIO and Hydra extend algorithm portfolio idea exploiting algorithm configuration techniques. In the literature, we can find further performing systems that combine portfolio and configuration methods, managing to win several solving competitions. For instance, 3S [22] combines algorithm selection with a static SAT solver schedule; a successive work augments its idea defining a dynamic parallel portfolio [29]. Two other famous successful systems are ISAC [23] and CSHC [30].

Claspfolio 2 [16] extends its previous version increasing the robustness of selection by defining a static pre-solving scheduling that may intervene if the learned selection model performs poorly. Given in input a set of ASP problem instances $\mathcal{I}$ and a portfolio of solvers $\mathcal{P}$, first of all, a training collection is built like in CLASPFOLIO with a pre-processing phase on both features and performances. The training phase consists of two parts: one computes a model $\mathcal{M}$ that maps each instance features and solver in $\mathcal{P}$ into a scoring value used to predict the best performing algorithm, while the other defines a pre-schedule of solvers, Pre, like ASPEED. According to user choice, $\mathcal{M}$ can be evaluated using cross-validation techniques: if the model performs particularly well, then the time slice allocated to Pre will be short or even null, otherwise, the scheduling will be defined considering the whole available time and $\mathcal{M}$ prediction gets limited consideration.

[7] ParamILS [21] contains two evaluation solutions: BasicILS and FocusedILS.

When a new problem instance i must be solved, its features x_i are extracted and used to select the best-predicted algorithm $\mathcal{A}$ using $\mathcal{M}$. If this process fails, a backup-solver is run, otherwise, the pre-solving schedule Pre is executed. If $\mathcal{A}$ is contained in Pre, it is removed from the scheduling and its time slice is distributed among the remaining solvers. If i has not been solved by the pre-schedule, then $\mathcal{A}$ runs for the remaining time.

AutoFolio A key aspect of CLASPFOLIO 2 is that implements several machine learning techniques and components that can be chosen to learn the portfolio model. AUTOFOLIO [27] performs *Algorithm Configuration* over CLASPFOLIO 2 framework, choosing the optimal learning techniques with its optimal parameters configuration. More precisely, four top-level parameters are considered: pre-solving, performance pre-processing, algorithm selector approach and features pre-processing, where each decision leads to further parameters setting. One crucial point is to evaluate the performance of models correctly; therefore, the system applies *model selection* cross-validation techniques.

Hydra [49] does not require a pre-defined portfolio: it takes in input a parametrized algorithm A, a training set of problem instances $\mathcal{I}$, and a performance metric m. Then it iteratively combines an automatic algorithm configurator $\mathcal{AC}$ and an algorithm selection system $\mathcal{AS}$ to build a dynamic portfolio of A's configurations.

Let $\mathcal{P}_k$ be the portfolio algorithm obtained at step k: at $k = 0$ it can be an empty set, or without loss of generality, we can already instantiate it with a set of algorithms. At each step, if $\mathcal{P}_k \neq \emptyset$, Hydra uses $\mathcal{AS}$ to learn a model that selects the most promising algorithm from $\mathcal{P}_k$ for each instance $i \in \mathcal{I}$, with respect to m. We indicate with $m(\mathcal{P}_k, i)$ the performance obtained by the selected algorithm in $\mathcal{P}_k$ over instance i. Then Hydra uses $\mathcal{AC}$ to find the best configuration of $\mathcal{A}$ that is able to improve the performance of the current portfolio $\mathcal{P}_k$. Given a candidate parameter configuration θ, a dynamic measure m_k is used to consider the performance of the portfolio built so far for each instance $i \in \mathcal{I}$:

$$m_k(\theta, i) = \max\{m(\theta, i), m(\mathcal{P}_k, i)\}.$$

Leveraging m_k, $\mathcal{AC}$ is able to find the parameter configuration θ_k that maximizes the portfolio performance and then it adds it to the set $\mathcal{P}_{k+1} = \mathcal{P}_k \cup \{\theta_k\}$. It iterates until a fixed number of steps or a time limit is reached.

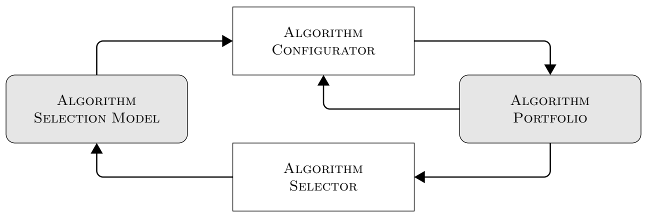

Fig. 3. Diagram of Hydra framework.

6 Conclusions

In recent years, *Algorithm Portfolio* systems are attracting widespread interest due to their outstanding performance. However, their exceptional results depend on the chosen portfolio solvers, as their strengths must be complementary with respect to the considered instances distribution. *Algorithm Configuration* can be leveraged to extend portfolio systems in several different ways. Since many modern systems are inspired by the previous, we analyzed some of the most influential works showing how their ideas and approaches evolve. In order to reproduce and thus compare the results of the systems, [8] introduces a standardized format and a repository called Algorithm Selection Library (ASlib). Moreover, the AS competitions 2015-2017 [28] contains comparison between different approaches.

Acknowledgments. This is a summary of my Master Double Degree thesis with professor Agostino Dovier as supervisor from the University of Udine and professor Martin Gebser from the Alpen-Adria Universität Klagenfurt. This work is partially supported by projects GNCS 2017-2019 and PRID UNIUD ENCASE.

References

1. A. Aamodt and E. Plaza. Case-based reasoning; foundational issues, methodological variations, and system approaches. *AI COMMUNICATIONS*, 7(1):39–59, 1994.
2. R. Amadini, M. Gabbrielli, and J. Mauro. Sunny: a lazy portfolio approach for constraint solving. *Theory and Practice of Logic Programming*, 14(4-5):509–524, 2014.
3. R. Amadini, M. Gabbrielli, and J. Mauro. A multicore tool for constraint solving. In *IJCAI*, pages 232–238, 2015.
4. R. Amadini, M. Gabbrielli, and J. Mauro. Why cp portfolio solvers are (under) utilized? issues and challenges. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 349–364. Springer, 2015.
5. C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *International Conference on Principles and Practice of Constraint Programming*, pages 142–157. Springer, 2009.
6. P. Balaprakash, M. Birattari, and T. Stützle. Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. In *International workshop on hybrid metaheuristics*, pages 108–122. Springer, 2007.
7. M. Balduccini. Learning and using domain-specific heuristics in asp solvers. *AI Communications*, 24(2):147–164, 2011.
8. B. Bischl, P. Kerschke, L. Kotthoff, M. Lindauer, Y. Malitsky, A. Fréchette, H. Hoos, F. Hutter, K. Leyton-Brown, K. Tierney, et al. Aslib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, 2016.
9. M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. T. Schneider, and S. Ziller. A portfolio solver for answer set programming: Preliminary report. In *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, pages 352–357. Springer, 2011.
10. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *IJCAI*, volume 7, pages 386–392, 2007.
11. M. Gebser, T. Schaub, and S. Thiele. Gringo: A new grounder for answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 266–271. Springer, 2007.

12. M. Gelfond and Y. Kahl. *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press, 2014.
13. C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.
14. T. Hastie, R. Tibshirani, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition*. Springer series in statistics. Springer, 2009.
15. H. Hoos, R. Kaminski, M. Lindauer, and T. Schaub. aspeed: Solver scheduling via answer set programming. *Theory and Practice of Logic Programming*, 15(1):117–142, 2015.
16. H. Hoos, M. Lindauer, and T. Schaub. claspfolio 2: Advances in algorithm selection for answer set programming. *Theory and Practice of Logic Programming*, 14(4-5):569–585, 2014.
17. B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.
18. F. Hutter, H. Hoos, and K. Leyton-Brown. Bayesian optimization with censored response data. *CoRR*, 2013.
19. F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.
20. F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.
21. F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *AAAI*, volume 7, pages 1152–1157, 2007.
22. S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm selection and scheduling. In *International Conference on Principles and Practice of Constraint Programming*, pages 454–469. Springer, 2011.
23. S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. Isac-instance-specific algorithm configuration. In *ECAI*, volume 215, pages 751–756, 2010.
24. P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann. Automated algorithm selection: Survey and perspectives. *CoRR*, abs/1811.11597, 2018.
25. L. Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data Mining and Constraint Programming*, pages 149–190. Springer, 2016.
26. K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. A portfolio approach to algorithm select. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 1542–1543, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.
27. M. Lindauer, H. H. Hoos, F. Hutter, and T. Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015.
28. M. Lindauer, J. N. van Rijn, and L. Kotthoff. The algorithm selection competition series 2015-17. *arXiv preprint arXiv:1805.01214*, 2018.
29. Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Parallel sat solver selection and scheduling. In *International Conference on Principles and Practice of Constraint Programming*, pages 512–526. Springer, 2012.
30. Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm portfolios based on cost-sensitive hierarchical clustering. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

31. M. Maratea, L. Pulina, and F. Ricca. Applying machine learning techniques to asp solving. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 17. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.
32. M. Maratea, L. Pulina, and F. Ricca. A multi-engine approach to answer-set programming. *Theory and Practice of Logic Programming*, 14(6):841–868, 2014.
33. O. Maron and A. W. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In *Advances in neural information processing systems*, pages 59–66, 1994.
34. J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*, 2009.
35. J. P. Marques Silva and K. A. Sakallah. Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, 1996.
36. T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.
37. E. Nudelman, K. Leyton-Brown, A. Devkar, Y. Shoham, and H. Hoos. Satzilla: An algorithm portfolio for sat. *Solver description, SAT competition*, 2004, 2004.
38. E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish conference on artificial intelligence and cognitive science*, pages 210–216, 2008.
39. C. H. Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.
40. L. Pulina and A. Tacchella. A multi-engine solver for quantified boolean formulas. In *International Conference on Principles and Practice of Constraint Programming*, pages 574–589. Springer, 2007.
41. S. Raschka. Model evaluation, model selection, and algorithm selection in machine learning. *CoRR*, abs/1811.12808, 2018.
42. J. R. Rice. The algorithm selection problem. *Advances in computers*, 15:65–118, 1976.
43. F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.
44. O. Roussel. Description of ppfolio (2011). *Proc. SAT Challenge*, page 46, 2012.
45. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.
46. K. A. Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys (CSUR)*, 41(1):6, 2009.
47. D. H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996.
48. D. H. Wolpert, W. G. Macready, et al. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.
49. L. Xu, H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *AAAI*, volume 10, pages 210–216, 2010.
50. L. Xu, H. H. Hoos, and K. Leyton-Brown. Hierarchical hardness models for sat. In *International Conference on Principles and Practice of Constraint Programming*, pages 696–711. Springer, 2007.
51. L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla-07: the design and analysis of an algorithm portfolio for sat. In *International Conference on Principles and Practice of Constraint Programming*, pages 712–727. Springer, 2007.
52. L. Xu, F. Hutter, J. Shen, H. H. Hoos, and K. Leyton-Brown. Satzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge*, pages 57–58, 2012.