Instance-Aware AMOSUM Propagator Selection in ASP*

Mario Alviano[1], Carmine Dodaro[1], Thomas Eiter[2], Salvatore Fiorentino[1,*], Tobias Geibinger[2] and Megan Macrì[1]

[1]Department of Mathematics and Computer Science, University of Calabria
[2]Knowledge-based Systems Group, Institue of Logic and Computation, TU Wien

Abstract

Answer Set Programming (ASP) is a well-known declarative paradigms for knowledge representation and automated reasoning, where solutions are represented as stable models of logic programs. The success of ASP is due to a rich set of language extensions that ease the modeling of complex domains, such as *sum* aggregates. A common pattern in real-world problems is the co-occurrence of *sum* constraints with *At-Most-One* (AMO) constraints. In particular, *sum* constraints assert that a weighted sum of literals meets a given threshold, while *amo* constraints restrict the simultaneous truth of multiple elements in a set.

Recently, the AMOSUM propagator has been introduced to jointly handle *sum* and AMO constraints, demonstrating significant performance improvements compared to approaches that treat these constraint types independently. This propagator has been further investigated in ongoing work,proposing several configurations that enhance the original AMOSUM propagator. However, no single configuration has been shown to dominate across all problem instances, motivating the need for a more adaptive strategy.

In this work, we propose a Machine Learning (ML) based approach to automatically select the most effective propagator configuration on a per-instance basis. We describe the construction of a diverse training dataset across three combinatorial benchmarks and we extract from it semantically meaningful features capturing the interaction between SUM and AMO constraints. Finally, we develop a series of ML-based selectors to obtain best propagator configurations based on the extracted features. Experimental results show that the ML-based selectors substantially outperform any single static configuration in terms of both solved instances and average runtime.

Keywords

Answer Set Programming, Logic Programming, Constraints

1. Introduction

Answer Set Programming (ASP) [1, 2] is a widely-adopted paradigm for knowledge representation and automated reasoning, thanks to its declarative nature and its ability to concisely model complex combinatorial problems [3, 4]. During the years, the plain language of ASP has been extended with several constructs aimed at improving the expressiveness, flexibility, and modeling capabilities of the language. Among these, aggregate constructs, particularly SUM constraints, play a central role in practical modeling [5]. A SUM constraint of the form

$$\mathrm{SUM}\{w_1 : \ell_1; \ldots; w_n : \ell_n\} \geq b$$

requires that the sum of the weights associated with true literals among $\ell_1, \ldots \ell_n$ meets a given threshold.

Besides their modeling relevance, aggregates also have a significant impact on solving efficiency. Modern ASP solvers rely on propagation mechanisms to infer consequences of partial assignments as early as possible, thereby reducing the search space during search. To this end, dedicated aggregate propagators exploit the semantics of SUM constraints to derive assignments that are necessary to satisfy

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026) co-located with the Federated Logic Conference (FLoC 2026).

*Mario Alviano, Carmine Dodaro, Thomas Eiter, Salvatore Fiorentino, Tobias Geibinger, Megan Macrì
mario.alviano@unical.it (M. Alviano); carmine.dodaro@unical.it (C. Dodaro); thomas.eiter@tuwien.ac.at (T. Eiter); salvatore.fiorentino@unical.it (S. Fiorentino); tobias.geibinger@tuwien.ac.at (T. Geibinger); mcrmgn02l65d122a@studenti.unical.it (M. Macrì)
0000-0002-2052-2063 (M. Alviano); 0000-0002-5617-5286 (C. Dodaro); 0000-0001-6003-6345 (T. Eiter); 0009-0004-6531-4139 (S. Fiorentino); 0000-0002-0856-7162 (T. Geibinger)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-03

CEUR-WS.org/Vol-4311/ASPOCP_regular_11.pdf

the corresponding inequality, often achieving substantially stronger pruning than purely propositional encodings.

In many real-world applications, the literals involved in SUM constraints are also subject to structural restrictions such as at-most-one (AMO) constraints, which limit the number of simultaneously true elements within a set. When these constraints are handled independently, standard SUM propagators fail to exploit this additional structure, leading to weaker inference and reduced efficiency.

To address this limitation, the AMOSUM construct has been introduced, enabling a tighter integration of SUM constraints with AMO conditions. Originally proposed by Alviano et al. [6], the construct has been further extended in ongoing work, introducing additional features such as reason minimization and alternative propagation strategies, including lazy variants. These developments result in multiple propagator configurations, each offering different trade-offs in terms of propagation strength and computational overhead. However, no single configuration consistently outperforms the others across all problem instances, highlighting the need for adaptive techniques that can select the most suitable configuration depending on the instance at hand.

In recent years, machine learning has proven effective for algorithm selection, where the goal is to automatically choose the best solver or configuration for a given problem instance. Approaches such as ME-ASP [7] have demonstrated that instance-based selection guided by cheaply computable features can significantly improve overall performance compared to any fixed strategy.

In this work, we adopt and specialize this idea to the context of AMOSUM propagation. We propose a machine learning-based approach that, given an unseen instance, automatically selects the most effective propagator configuration. Our method relies on a diverse training dataset constructed from multiple combinatorial benchmarks and on the extraction of features that capture relevant properties of the interaction between SUM and AMO constraints. Based on these features, we train classifiers that predict the best configuration to use for each instance.

Experimental results show that the proposed approach consistently outperforms any single static configuration, both in terms of the number of solved instances and average runtime. These findings highlight the potential of combining advanced propagation techniques with data-driven selection strategies in ASP.

2. Preliminaries

This section briefly recaps the AMOSUM construct [6] and the additional configurations proposed in ongoing work, preceded by the necessary background on ASP and aggregate propagation.

2.1. Answer Set Programming

An ASP program consists of rules of the form $p \leftarrow \ell_1, \ldots, \ell_n$, where each ℓ_i is either a positive atom a or its default negation $\sim a$. The complement of a literal is defined as $\overline{a} = \sim a$ and $\overline{\sim a} = a$. Aggregate constraints such as $\text{SUM}\{w_1 : \ell_1;\ \cdots\ ;\ w_n : \ell_n\} \geq b$, requiring the weighted sum of true literals among $\ell_1,\ldots, \ell_n$ to meet a bound b. Solutions are computed as *stable models* [8], i.e., minimal models of the reduct with respect to an interpretation.

Modern ASP solvers find stable models via *conflict-driven clause learning* (CDCL) [9, 10], iterating over a *choose–propagate–learn* loop. At each step, a literal is heuristically chosen and added to the current assignment I; a *propagator* then derives any deterministic consequences, each recorded with a *reason* clause. When a contradiction arises, backward resolution on the conflicting reasons produces a new learned clause, and search backtracks. For SUM constraints, a dedicated *aggregate propagator* [11, 12] infers a literal ℓ_i as true whenever omitting it would make the bound b unreachable given the current assignment. The conciseness of the reason provided at each inference step directly affects the quality of the learned clauses and, consequently, the efficiency of the search.

2.2. The AMOSUM construct

An AMOSUM constraint σ extends a SUM constraint by partitioning its literals into groups $s_1, \ldots, s_k$, each subject to an AMO condition: at most one literal per group may be true. Formally, σ has the form

$$\text{AMOSUM}\{w_1 : \ell_1\ [s_1];\ \cdots\ ;\ w_n : \ell_n\ [s_n]\} \geq b \tag{1}$$

where each literal ℓ_i belongs to part s_i, denoted $part_\sigma(\ell_i) := s_i$; by $lits_\sigma|_s$ we denote the set of literals in part s and by bnd_σ the bound b of the constraint σ. An interpretation I satisfies σ if the weighted sum of true literals meets b and at most one literal is true in each part.

Example 2.1. Let Π_{run} be the following program:

$$\begin{array}{rll} r_\alpha: & \alpha \leftarrow \sim\alpha' & \alpha \in \{x,y,z\} \\ r_{\alpha'}: & \alpha' \leftarrow \sim\alpha & \alpha \in \{x,y,z\} \\ \sigma_1: & \text{SUM}\{1:\sim x;\ 1:\sim y\ \} \geq 1 & \\ \sigma_2: & \text{SUM}\{1:\ x;\ 1:\ y;\ 2:z\} \geq 3 & \end{array}$$

Π_{run} can be rewritten with AMOSUM constraint by replacing σ_1 and σ_2 with

$$\sigma_3: \quad \text{AMOSUM}\{1:x\ [1];\ 1:y\ [1];\ 2:z\ [2]\} \geq 3$$

where $parts_{\sigma_3} = \{1,2\}$, $lits_{\sigma_3}|_1 = \{x,y\}$, $lits_{\sigma_3}|_2 = \{z\}$, and $bnd_\sigma = 3$. The first group arises from the AMO constraint σ_1, while z forms its own singleton group since no AMO constraint involves it. Note that the rewriting is currently performed manually.

A key quantity for propagation is the *maximum possible sum* $mps_\sigma(I)$: the highest weighted sum achievable by any total extension of I that respects all AMO constraints of σ. Intuitively, it represents how close the current assignment is to satisfying the bound. Given a literal ℓ, we also write $mps_\sigma(I,\ell)$ for the maximum possible sum assuming ℓ is true, and mps as shorthand for $mps_\sigma(\emptyset)$.

2.3. Propagation

A constraint σ of the form (1) is associated with two inference rules. In both cases, we assume an interpretation I consistent with the *at most one* constraint, i.e., $\sum_{\ell \in lits_\sigma(s)} I(\ell) \leq 1$ for all $s \in parts_\sigma$, where $I(\ell) = 1$ if ℓ is true with respect to I, and 0 otherwise.

- **AMO Rule (AMR).** The literal $\overline{\ell}$ is added to I if there exists $\ell' \in lits_\sigma|_{part_\sigma(\ell)}$ such that $\ell' \in I$. The corresponding reason is $reason(\overline{\ell}) = \{\overline{\ell}, \overline{\ell'}\}$.
- **Amosum Rule (ASR).** An undefined literal $\ell \in lits_\sigma$ is inferred true when assuming it false would cause the *max possible sum* to fall below the bound. Formally, ℓ is added to I if J is the first prefix (subset) of I such that $mps_\sigma(J,\overline{\ell}) < bnd_\sigma$. where $mps_\sigma(J,\overline{\ell})$ is the maximum possible sum assuming ℓ false, i.e., $mps_\sigma(J,\overline{\ell}) := \sum_{s \in parts} mwh_\sigma(J \dot\cup \{\overline{\ell}\}, s)$ with

$$mwh_\sigma(J,s) := \begin{cases} w_{\ell'} & \text{if } lits_\sigma|_s \cap J = \{\ell'\} \\ \max\{w \cdot J^\uparrow(\ell') \mid (w:\ell'\ [s]) \in \sigma\} & \text{otherwise} \end{cases}$$

and $J^\uparrow(\ell) = 1$ if ℓ is not false with respect to J, and 0 otherwise.
The corresponding reason is defined by all literals that either (1) could contribute to the max possible sum or (2) are true in a parition, given that the current mps is entirely defined by their truth values.

Reason Minimization. Propagators use reasons to explain deterministic consequences. In general, a more concise reason leads to stronger learned clauses during conflict analysis, reducing the search space more effectively. Therefore, to improve performance, a reason minimization algorithm has been implemented, based on an *increment* function $\text{inc}(R,\ell,\ell')$ that quantifies the increase in mps achievable by removing literal ℓ' from $reason(\ell)$. The minimization is performed *on-the-fly* during reason construction, keeping its complexity polynomial. A literal is considered unessential if its removal does not push the mps above the threshold bound.

Propagation Laziness. For instances where the aggregate constraint remains far from its bound, invoking the propagator at each assignment step incurs unnecessary overhead. This is addressed by a *laziness parameter* $p \in [0, 1]$: propagation is triggered only when $b/\mathrm{mps}_\sigma(I) \geq p$. The most effective one has been shown to be the *full lazy* one, that is, the one adopting $p = 1$.

Note. The last two contributions have already been implemented and are currently under review as part of an extended journal version. Their implementation is publicly available in the GitHub repository referenced at the end of the paper.

2.4. ML-based selectors

To automatically select the best propagator configuration for a given instance, we formulate the problem as a classification task: given a feature vector describing an instance, predict the configuration expected to perform best, where best is defined as the configuration obtaining the lowest wall-clock solving time on that instance. Here, a configuration refers to a specific instantiation of the AMOSUM propagator, defined by the choice of propagation strategy (eager or lazy) and the activation of reason minimisation. Different configurations expose different trade-offs between inference strength and computational overhead, and no single one dominates across all problem instances. We now describe the considered classification algorithms.

KNN. A non-parametric method that classifies a new instance by majority vote among its k nearest neighbours in feature space. It makes no assumption about the underlying feature distribution and works well when similar instances tend to require the same solution strategy [13].

FOLD-R++. A lightweight declarative rule induction algorithm that learns a set of default rules with exceptions, producing human-interpretable if-then classifiers. Given a set of labelled examples, it iteratively induces rules that cover positive instances while minimising false positives, controlled by a `ratio` parameter that balances positive and negative coverage during rule induction. In a multi-class setting, it is applied via a one-vs-rest decomposition, training one rule set per class [14].

Decision Trees and Forests. Single decision trees build a hierarchical partition of the feature space through threshold tests on individual features. They are fast and interpretable; a tree of depth d requires at most d feature comparisons to reach a prediction, but may sacrifice accuracy when decision boundaries are complex [15]. *Random Forests* extend this by ensembling multiple trees trained on random feature subsets, reducing variance and producing more stable predictions [16]. *XGBoost* further improves on this through gradient boosting, where each tree corrects the residual errors of its predecessors; built-in class weighting, subsampling, and L1/L2 regularization make it robust on large datasets [17]. *LightGBM* is a variant of gradient boosting that uses histogram-based splitting and leaf-wise tree growth, producing deeper and more specialized trees that capture fine-grained patterns at lower computational cost [18].

3. Dataset Generation and Feature Extraction

This section describes the experimental infrastructure used to train and evaluate the ML-based solver selection framework. We first detail the dataset generation process, in which three classical combinatorial benchmarks were extended to increase structural diversity and amplify performance variance across propagation strategies. We then describe the feature extraction pipeline, motivating the introduction of two new families of constraint-specific features tailored to AMOSUM aggregates, which complement the structural features of the ME-ASP framework by capturing the semantic interaction between group weights and logical thresholds.

3.1. Dataset Generation

Three classical combinatorial benchmarks were selected and specifically modified to maximise variance across the dimensions that most influence the performance gap between propagation strategies.

Knapsack (KC). The Knapsack problem requires selecting items, each with a weight and a value, to achieve a minimum total value without exceeding a capacity bound. An AMO constraint restricts selection to at most one variant per item category, making the AMOSUM formulation directly applicable. The instances presented in [6] were further extended. First, *scenario heterogeneity* was introduced by defining three weight/value regimes (small, medium, and large items), varying constraint tightness and thus testing mps-based inference across a range of difficulty levels. Second, *difficulty shifting* was applied: rather than distributing instances evenly across satisfiability categories, the instance generator assigns a 40% probability to instances near the SAT/UNSAT phase transition, where the performance gap between propagation strategies is most pronounced. This yields 1800 instances in total.

Group Assignment (GA). The Group Assignment benchmark models a workforce allocation problem in which each person must be assigned working hours to projects within monthly limits (an AMO constraint per person), while guaranteeing a minimum budget expenditure per project (a SUM constraint over hourly salaries). This naturally maps to an AMOSUM formulation. A dense systematic grid was employed, with the number of projects ranging from 5 to 40 and personnel from 50 to 300. Instance generation was intentionally unbalanced, assigning higher weight to borderline satisfiability categories (10 instances each for *middle* and *possibly_unsat* vs. 5 for clearly SAT or UNSAT), as these regions are where the difference between lazy and eager propagation is most pronounced. This yields 1440 instances in total.

Weighted Graph Colouring (WGC). In the WGC problem, at most one colour is assigned to each graph node (the AMO constraint). Colours carry exponentially growing weights (e.g., red: 2, green: 4, blue: 8, yellow: 16, cyan: 64), and a global SUM constraint requires the total assigned weight to exceed a threshold. Rather than relying on the small fixed set of ASP competition instances, as in [6], random graphs were synthesised using the Erdős-Rényi $G(n, p)$ model. A parametric grid was constructed by varying node count $N \in \{20, 40, 60, 80, 100\}$ and edge density $d \in \{0.3, 0.5, 0.7\}$, generating 10 base graphs per (N, d) pair. The threshold grid was expanded to 9 points in the range $[0.10, 0.90]$ of the theoretical maximum to thoroughly map the decision boundary. Finally, uniform noise of $\pm 2\%$ was injected into threshold values to prevent classifiers from memorizing exact numerical thresholds and to encourage learning of underlying structural relationships. This yields 1350 instances in total.

Total Dataset and External Data Integration. The combined generation across the three benchmarks yields: 1800 (KC) $+ 1440$ (GA) $+ 1350$ (WGC) $= 4590$ total instances. To further improve generalization, the generated instances were merged with instances from preliminary work, and the unified dataset was split in an 80/20 ratio for training and testing. The structural variance resulting from grid expansion, multiple benchmarks, and noise injection provides a solid foundation against overfitting.

Dataset Proprocessing. Raw performance data from benchmark runs contains substantial noise. Two filtering criteria were applied to ensure the training signal reflects genuine algorithmic differences:

- `min_time_relevance` **(0.5 s):** Differences below half a second are treated as OS noise and excluded. This prevents the model from learning spurious rules based on hardware fluctuations.
- `min_gap_value` **(20%):** An instance is considered significant for training only if the runner-up configuration is at least 20% slower than the winner. Near-ties generate training noise that degrades predictions on high-stakes cases where the correct choice matters.

Instances not solved within the time limit (see below) by any configuration were completely excluded. After filtering, the AMO dataset contained 1580 valid instances (1264 train / 316 test).

Table 1
AMO-specific features for ML solver selection.

Feature	Significance
b_Σ	Sum of bounds over all AMOSUM constraints.
mps_Σ	Sum of Maximum Possible Sums over all AMOSUM constraints.
W_Σ	Sum of naive total weight sums over all AMOSUM constraints.
mps_ρ	b_Σ/mps_Σ: proximity of the threshold to the AMO-aware ceiling.
ρ_Σ	mps_Σ/W_Σ: theoretical gain of AMOSUM over SUM.
τ	b_Σ/W_Σ: constraint severity relative to total weight.

3.2. Feature Extraction for ML-Based Solver Selection

We complement the global syntactic analysis of the program with a dedicated characterisation of its aggregate properties, motivating why standard ASP descriptors are insufficient for capturing the internal logic of AMOSUM constraints and formally defining the new metrics used to map the interaction between group weights and logical bounds.

Limitations of Structural Features. The initial selector design relied exclusively on syntactic features from the ME-ASP framework [7]: number of rules, atoms, fractions of unary/binary/Horn rules, and related structural statistics. These features are cheap to compute and effective for general ASP solver selection, but exhibit a critical limitation in the context of parametric AMOSUM instances.

Consider two WGC instances derived from the same base graph but with different threshold values (e.g., 10% vs. 90% of the theoretical maximum). After grounding, both produce identical structural feature vectors, since the threshold is encoded as an integer literal in a fact and does not affect rule or atom counts. This causes two interrelated problems, i.e., the *label aliasing* as the classifier receives identical input vectors associated with different optimal labels, making it impossible to learn a consistent decision boundary; and the *semantic unawareness* as structural features fail to reflect how strongly a constraint restricts the search space, failing to characterize when AMOSUM-based propagation provides a meaningful advantage. This motivates the introduction of AMOSUM-specific features that directly measure the logical interaction between constraint thresholds and partition weights.

SUM-AMO Interaction Features. The full set of AMO-specific features is detailed in Table 1. The aggregated features b_Σ, mps_Σ, and W_Σ capture the overall scale of the problem by summing bounds, maximum possible sums, and naive weight sums across all AMOSUM constraints, respectively. Their magnitude provides a coarse measure of instance difficulty.

The ratio $mps_\rho = b_\Sigma/mps_\Sigma$ measures how close the total bound is to the AMO-aware ceiling. High values (close to 1) characterise tight constraints where propagation fires frequently and eager strategies are typically more effective; low values indicate looser constraints where lazy propagation may reduce unnecessary overhead.

The ratio $\rho_\Sigma = mps_\Sigma/W_\Sigma$ quantifies the theoretical advantage of AMOSUM over a plain SUM propagator. Values close to 0 indicate that the AMO structure substantially reduces the maximum reachable sum relative to the naive total, so the AMOSUM propagator operates much closer to the bound and triggers earlier inference. Values close to 1 suggest that the AMO information provides little additional benefit, yielding behaviour comparable to a plain SUM constraint.

Finally, $\tau = b_\Sigma/W_\Sigma$ measures overall constraint tightness with respect to the naive weight sum, providing a baseline difficulty signal that is agnostic to AMO structure.

Filtered ME-ASP Features. From the full ME-ASP feature set, we retain only those that proved most relevant in our experiments, that are the number of atoms in the ground program ($atoms$), the number

of ground rules (r), the number of ground facts (f), the proportion of Horn rules among ground rules ($horn_\rho$), the proportion of unary rules among ground rules ($unary_\rho$), and the proportion of binary rules among ground rules ($binary_\rho$).

4. Implementation and Experiments

The Implementation and Experiments section aims to validate the effectiveness of the Machine Learning-based selector and to analyse the tested architectures in detail. The core of this section is the presentation of the best-performing model, structured as a two-stage cascade classification pipeline, in which a lightweight KNN Triage model handles the majority of instances, delegating uncertain cases to a Deep Random Forest Oracle.

To validate the superiority of this approach, the section presents an ablation study conducted on the test set, where the individual models are evaluated in isolation to measure their specific contributions and compare their stand-alone performance against the complete pipeline. Finally, a subsection is dedicated to discussing the alternative models evaluated during the development phase (such as XGBoost). While these models proved to be robust and performed well, the section will explain why they fell short of the optimal balance between accuracy and inference time offered by the cascade architecture.

4.1. Implementation

The empirical validation was carried out using Python. The final dataset, constructed from the three combinatorial benchmarks described in Section 3, was partitioned into training (80%) and testing (20%) sets using a stratified split to preserve the original distribution of winning configurations across both partitions.

Algorithm selection datasets naturally exhibit class imbalance. To prevent models from simply predicting the most frequent winning configuration and ignoring under-represented but highly specialised ones, balanced sample weights were applied during training. A domain-specific penalty was additionally introduced: the sample weights of instances where AMOCLINGO-L was the winning configuration were multiplied by a factor of 1.8. This adjustment penalises the classifier more heavily for misclassifying lazy-optimal instances, a critical scenario where deploying an eager propagator on a loose constraint would incur massive and unnecessary computational overhead.

To identify the optimal cascade architecture, a custom AutoML grid search was employed. During training, all Level 1 and Level 2 candidate models are trained independently on the full training set. At inference time, the cascade operates via a *confidence-rejection mechanism*: an instance is first evaluated by the lightweight Level 1 (Triage) model; if the maximum predicted class probability meets or exceeds a predefined confidence threshold, the prediction is accepted immediately. Otherwise, the instance is delegated to the heavier Level 2 (Oracle) model.

It should be noted that model selection was performed directly on the held-out test set, which may introduce a mild optimistic bias in the reported figures. However, the impact of this choice is considered limited: the search space is structured and relatively small, consisting of a fixed set of standard classifiers combined with a coarse grid of four confidence thresholds. Furthermore, the Level 1 Triage models were deliberately chosen from low-complexity families, whose limited memorisation capacity acts as a structural regulariser for the cascade as a whole. To provide empirical support for these claims, the winning architecture was subsequently validated via 10-fold stratified cross-validation on the training partition exclusively, yielding an accuracy of $87\% \pm 2.32\%$. The low standard deviation across folds confirms that the pipeline generalises stably across different data splits and that the reported test-set figures are not attributable solely to a favourable random partition.

The search space explored the following combinations:

- **Triage Candidates (Level 1):** Logistic Regression, Decision Trees (depth 4 and 6), Shallow Random Forests (depth 5), KNN (k=5, uniform weights), Linear support vector machine (SVM), and a custom rule-based FoldR++ wrapper with one-vs-rest decomposition.

- **Oracle Candidates (Level 2):** Deep Random Forests, XGBoost, LightGBM, Distance-weighted KNN (k=5), and Radial Basis Function (RBF) kernel SVM.
- **Threshold Grid:** Confidence thresholds of 0.65, 0.75, 0.85, and 0.90 were evaluated for all pairs, with the exception of `FoldR++`, for which only a fixed threshold of 0.75 was used, given the binary nature of its probability outputs.

The search identified **KNN (k=5)** as the optimal Triage layer and a **Deep Random Forest** as the Oracle, with a confidence threshold of **0.65**, as the best-performing combination. Crucially, the Triage layer autonomously resolved 81.6% of the instances, meaning the computationally expensive Deep Random Forest was invoked for fewer than 20% of cases, demonstrating that the cascade drastically reduces classification overhead while maintaining high accuracy.

4.2. Selection Algorithms

Algorithm selection for ASP propagators presents challenges that standard supervised learning benchmarks do not: moderate dataset size with deliberate class imbalance, noisy labels from solver stochasticity, and severe misclassification asymmetry, routing a hard instance to the wrong solver can waste orders of magnitude more computation than a correct prediction saves. The following classifiers were evaluated with these constraints in mind.

KNN. Its suitability here stems from the meaningful geometry of the feature space: instances with similar $\mathrm{mps_ratio}$, $\mathrm{tightness}$, and ρ genuinely tend to require the same propagation strategy. The AutoML cascade exploits this by using KNN as the triage layer for AMO constraints, processing 81.6% of test instances with high confidence at near-zero inference cost.

Decision Trees and Ensembles. Single trees offer speed and interpretability, depth d requires at most d feature comparisons, but shallow variants sacrifice accuracy on the complex boundaries separating AMOSUM configurations. *Random Forests* recover stability through bagging and random feature subsets; deep variants act as the Oracle in the final cascade, where the ensemble's non-linear decision regions better capture the tight geometric boundaries between AMOCLINGO-INF-MR and AMOCLINGO-L in the mps_ρ–τ feature space.

Gradient Boosting (XGBoost and LightGBM). XGBoost was evaluated as a primary Oracle candidate. Its residual-correction mechanism, built-in class weighting, and L1/L2 regularization make it robust on the ∼1200–1500 training instances available and naturally suited to tabular feature vectors. However, the AutoML search ultimately selected a Deep Random Forest as the Oracle for the AMOSUM cascade, as its non-linear ensemble regions better capture the tight geometric boundaries separating AMOCLINGO-INF-MR from AMOCLINGO-L in the feature space. LightGBM, while competitive, offered no measurable advantage over the Random Forest on the AMOSUM feature set, whose relatively low dimensionality does not benefit from the finer-grained histogram-based partitioning that makes LightGBM preferable on richer feature spaces.

FOLD-R++. In our setting, it is wrapped in a one-vs-rest decomposition to handle the three-class target. The `ratio` parameter, which controls the balance between positive and negative coverage during rule induction, was tuned to 0.3 for the AMOSUM feature set. Although FOLD-R++ was ultimately not selected as the winning triage model, it proved competitive: its declarative rules can identify clearly lazy-optimal or clearly eager-optimal instances at near-zero cost, effectively pre-filtering a substantial fraction of the test set and delegating only genuinely ambiguous instances to the XGBoost Oracle. Since FOLD-R++ rules fire deterministically, the predicted confidence is always 0 or 1, making the confidence threshold a fixed parameter rather than a tuneable one. This prevents the cascade from exploiting the full threshold grid and reduces routing flexibility compared to probabilistic classifiers such as KNN or Random Forests.

Dataset Formalization and Target Labels. The final dataset consists of instances represented by the feature vectors extracted in Section 3.2. Each instance is mapped to a label identifying the most effective AMOSUM propagator configuration on that instance. Labels were obtained empirically: each instance was executed against all candidate propagator configurations under a fixed time limit of 1200s, and the configuration achieving the lowest wall-clock solving time was assigned as the ground-truth label. In case of a timeout, the instance was excluded from the training set. The classification task thus aims to learn a mapping from the combined structural and semantic feature space to one of the following three target configurations:

- AMOCLINGO-INF-MR: eager propagation with reason minimisation, preferred on tight instances where the bound is close to the AMO-aware ceiling and the propagator fires frequently.
- AMOCLINGO-L: full lazy propagation, preferred on loose instances where eager inference would fire rarely and incur unnecessary overhead.
- CLINGO: the baseline CLINGO solver with standard AMO handling, competitive on instances where the AMOSUM structure provides little additional propagation benefit (i.e., high ρ_{Σ}).

By pairing the AMOSUM-specific features, particularly mps_{ρ}, ρ_{Σ}, and τ, with these performance labels, the selector learns to bypass the label aliasing problem described in Section 3.2 and identify the most efficient propagation strategy on a per-instance basis.

Experimental Setup. All benchmark runs were conducted on a computing cluster managed via SLURM[1], consisting of 10 nodes , each equipped with two Intel Xeon Silver 4314 processors (16 cores per socket at 2.40 GHz, 24 MB L3 cache) and 512 GB of RAM, running Ubuntu 22.04 LTS (kernel 5.15.0-131-generic). To improve reproducibility and reduce non-deterministic factors, hyperthreading, ASLR, and NUMA were disabled; additionally, 2 cores per CPU were reserved for the scheduler and OS, and the L3 cache was restricted to 2 MB per pair of cores via Intel RDT. Each job was allocated 2 cores, with a maximum of 14 concurrent jobs per node. Each instance was executed against all candidate propagator configurations with a wall-clock time limit of 1200 seconds. Instances not solved by any configuration within this limit were excluded from the dataset, as described in Section 3.

Computational Cost of Preparation Steps. Generating the training labels required a cumulative 142 CPU days, parallelised across the cluster described above. The training step itself was fast: both KNN ($k = 5$) and Random Forest (300 trees) completed in under 15 seconds, with all training conducted on a GPU T4 $\times 2$. Feature extraction was performed using the CLINGO API to ground and inspect each instance as described in Section 3.2, requiring negligible time, while the dataset generation is the bottleneck, as all systems must be executed for every instance in the training set.

4.3. ML-based Selector: Design and Evaluation

To evaluate the effectiveness of the proposed ML-based selector, we compared its overall solving performance against the three static baseline configurations: CLINGO, AMOCLINGO-L, and AMOCLINGO-INF-MR. The primary evaluation metric is the total number of instances solved within the allocated time limit, complemented by the per-class classification metrics reported in Table 2.

Classification Performance. The cascade selector achieved an overall accuracy of **90.82%** on the 316 test instances, with 81.6% of them resolved directly by the KNN Triage layer without invoking the Oracle. Table 2 reports the full per-class breakdown.

The selector achieves its strongest performance on AMOCLINGO-INF-MR (F1 = 0.93), the most frequent class, which benefits from a well-populated region of the mps_{ρ}–ρ_{Σ} feature space. AMOCLINGO-L achieves a high recall of 0.94 but lower precision,reflecting the effect of the 1.8 $\times$ sample-weight penalty introduced during training: the model is successfully discouraged from routing lazy-optimal instances

[1] https://slurm.schedmd.com/

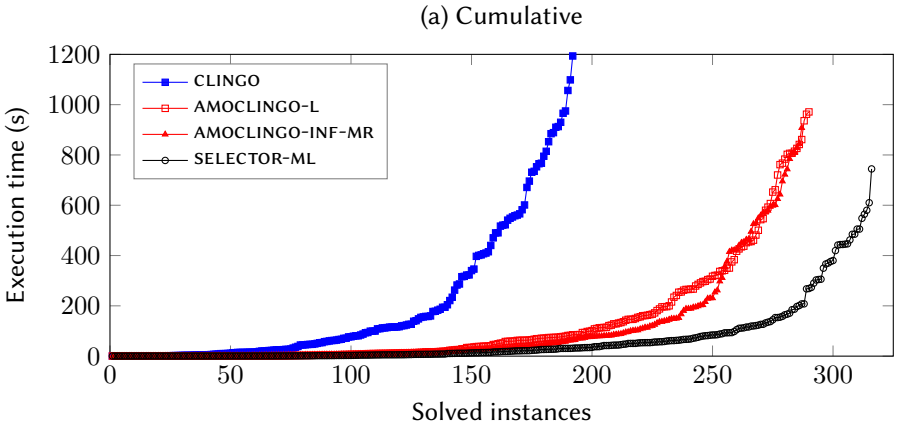


Figure 1: Cactus plot comparing the cumulative performance of all tested configurations (CLINGO, AMOCLINGO-L, AMOCLINGO-INF-MR, and SELECTOR-ML). The graph shows the execution time in seconds (y-axis) required to solve a given number of instances (x-axis).

Table 2
Per-class classification report of SELECTOR-ML on the AMO test set (316 instances).

Configuration	Precision	Recall	F1-score	Support
AMOCLINGO-INF-MR	0.94	0.91	0.93	158
AMOCLINGO-L	0.87	0.94	0.90	116
CLINGO	0.89	0.81	0.85	42
Weighted avg	0.91	0.91	0.91	316

to eager configurations, which would otherwise incur severe computational overhead. CLINGO shows the lowest F1 (0.85), providing a single balanced measure of classification accuracy. It is consistent with its smaller support (42 instances) and the geometric overlap it shares with AMOCLINGO-INF-MR in the high-ρ_Σ region, where AMOSUM provides limited propagation advantage over plain SUM.

Solving Performance. On the 316 test instances, SELECTOR-ML solves all instances. The 24 misclassified instances (error rate $\sim$7.6%) did not result in any timeouts, showing that, even when the predicted configuration is suboptimal, the performance degradation remains limited.

The cumulative performance of all configurations is summarised in the cactus plot (Figure 1), which plots the number of solved instances (x-axis) against cumulative execution time in seconds (y-axis). SELECTOR-ML clearly dominates all static baselines. It reaches the maximum of 316 solved instances while maintaining an exceptionally low cumulative execution time. The eager (AMOCLINGO-INF-MR) and lazy (AMOCLINGO-L) configurations perform comparably to each other, both plateauing at roughly 290 solved instances before experiencing a sharp increase in runtime, confirming that no single static configuration is universally better. The baseline CLINGO solver trails significantly, solving fewer than 200 instances within the time limit.

Feature Importance Analysis. To assess which features drive configuration selection, we analysed feature importance using the Gini index [15, 19] for the Deep Random Forest based Selector. The results show that both AMOSUM-specific and ME-ASP features contribute meaningfully to the decision process. The most influential feature is the sum of bounds b_Σ (Gini = 0.18), followed by the proportion of horn rules $horn_\rho$ (0.13) and the proportion of unary rules $unary_\rho$ (0.11). Among the remaining features, AMOSUM features prove more important overall: W_Σ and mps_Σ reach Gini values of 0.09, while the ME-ASP features $atoms$, f, $binary_\rho$, and r score 0.07, 0.05, 0.05, and 0.04, respectively. The AMOSUM ratio mps_ρ contributes least, with an importance of 0.01. Thus, the optimal configuration depends both

Table 3
Ablation study comparing standalone L1/L2 models against the final cascade selector on the AMO test set.

Model Layer	Algorithm	Accuracy	Misclassified	Total Solved
L1 Only	KNN	86.71%	37	315
L2 Only	Deep Random Forest	89.87%	26	**316**
Final Cascade	KNN + Deep RF	**90.82%**	**24**	**316**

on the structure of the AMOSUM aggregate and on general syntactic properties of the ASP program, such as the relative proportions of Horn and unary rules.

Ablation Study: Triage and Oracle vs. Cascade. To justify the added complexity of the two-stage cascade architecture, we conducted an ablation study comparing the performance of the standalone Level 1 (Triage) and Level 2 (Oracle) models against the final integrated cascade on the 316 test instances. The results are summarised in Table 3.

The cascade strictly improves upon both individual components across every metric. The standalone KNN Triage (L1) achieves 86.71% accuracy, misclassifying 37 instances and failing to solve 1 satisfiable instance within the time limit. Its weakness is concentrated on CLINGO, where the per-class F1 drops to 0.71 and recall falls to 0.60, indicating that the KNN struggles to correctly identify the high-ρ_{Σ} region where AMOSUM propagation provides little advantage over the plain SUM baseline. This is consistent with the geometric structure of the feature space: CLINGO-optimal instances occupy a sparse, poorly separated cluster that a non-parametric majority-vote classifier handles less reliably.

The standalone Deep Random Forest Oracle (L2) recovers substantially, reaching 89.87% accuracy and solving all 316 instances, with per-class F1 scores of 0.92, 0.89, and 0.84 for AMOCLINGO-INF-MR, AMOCLINGO-L, and CLINGO respectively. The ensemble's non-linear decision regions provide better coverage of the boundary between AMOCLINGO-INF-MR and AMOCLINGO-L in the mps_{ρ}–τ space, but applying it uniformly to all instances incurs unnecessary classification overhead on the 81.6% of cases that the KNN can already resolve confidently.

The final cascade achieves the best result across all metrics: 90.82% accuracy, only 24 misclassifications, and a perfect solving rate of 316/316. The marginal but consistent improvement over the standalone Oracle, confirms the synergistic effect of the confidence-rejection mechanism: by trusting the KNN on geometrically unambiguous instances and delegating only the borderline cases to the Deep Random Forest, the cascade avoids the noise introduced by routing easy instances through the heavier model. The per-class gains are modest but coherent, with the cascade improving F1 on both AMOCLINGO-INF-MR (0.93 vs. 0.92) and `amoclingo-l` (0.90 vs. 0.89), and matching the Oracle on CLINGO (0.85 vs. 0.84).

Comparison with Alternative Architectures To fully validate our final modelling choices, we compared the winning AutoML cascade architecture against alternative machine learning setups evaluated during the development phase. The goal is to understand the performance trade-offs across different algorithmic approaches, complexity levels, and interpretability requirements. All setups were evaluated on the same test set, measuring both the number of successfully solved instances and the average PAR1 solving time, where PAR1 is the average runtime penalising unsolved instances with the full timeout value.

We defined the following experimental configurations:

- **Experiment 1 (Flat XGBoost):** A single standalone XGBoost model representing a powerful but monolithic approach, processing every instance through the same gradient boosting pipeline without a triage mechanism.
- **Experiment 2 (Shallow Cascade):** A cascade prioritising interpretability and speed at the triage level, employing a very shallow Decision Tree (`max_depth = 3`) as L1 and XGBoost as L2, with a confidence threshold of 0.75.

Table 4
Performance comparison of ML architectures on the AMO dataset.

	KC (31 Instances)		WGC (105 Instances)		GA (180 Instances)	
Approach	**Avg PAR1 (s)**	**Solved**	**Avg PAR1 (s)**	**Solved**	**Avg PAR1 (s)**	**Solved**
Exp. 1 (Flat XGBoost)	110.42	**31**	**21.46**	**105**	**64.75**	**180**
Exp. 2 (Shallow Cascade)	308.01	26	34.07	**105**	136.82	179
Exp. 3 (Deep DT Cascade)	115.10	**31**	21.64	**105**	95.65	179
Exp. 4 (FOLD-R++ Cascade)	164.46	30	21.97	**105**	89.44	**180**
Exp. 5 (Final Cascade)	**93.88**	**31**	21.78	**105**	91.00	**180**
Exp. 6 (L1 Only — KNN)	150.74	30	21.62	**105**	92.17	**180**
Exp. 7 (L2 Only — Deep RF)	107.23	**31**	21.49	**105**	90.47	**180**

- **Experiment 3 (Deep DT Cascade):** A cascade using a depth-7 Decision Tree as L1 and a 300-estimator XGBoost as L2, with a confidence threshold of 0.65, attempting to balance accuracy with the structural transparency of a tree-based triage.
- **Experiment 4 (FOLD-R++ Cascade):** A cascade using FOLD-R++ as the L1 Triage model, a lightweight declarative rule extraction algorithm with `ratio = 0.3`, paired with a 300-estimator XGBoost Oracle and a confidence threshold of 0.75.
- **Experiments 5–7 (Final Cascade & Standalone Components):** The formally selected Final Cascade (Exp. 5) alongside its standalone L1 (Exp. 6) and L2 (Exp. 7) components, included for a complete domain-by-domain breakdown.

Results for AMO Constraints. The results reveal a hierarchy among the tested architectures. The Shallow Cascade (Exp. 2) performs poorly across all domains: its depth-3 Decision Tree cannot capture the geometric nuances of the mps_ρ–ρ_Σ feature space, leading to 5 unsolved KC instances and the highest PAR1 times overall. Increasing the tree depth in the Deep DT Cascade (Exp. 3) recovers accuracy on KC and WGC but at the cost of all interpretability, and still fails to solve 1 GA instance.

The Flat XGBoost (Exp. 1) is the strongest monolithic baseline, solving all instances in KC and GA with competitive PAR1 times. However, it applies the same heavy gradient boosting pipeline uniformly to every instance, including those that a lightweight model could resolve trivially. The FOLD-R++ Cascade (Exp. 4) demonstrates that a declarative, interpretable triage can effectively pre-filter instances, recovering most of Flat XGBoost's speed while retaining near-complete coverage.

The Final Cascade (Exp. 5) achieves the best overall profile: it is the only approach to solve all 316 instances across every domain while posting the lowest KC PAR1 time (**93.88s**), outperforming even the monolithic Flat XGBoost (110.42s) on the domain where constraint tightness varies most and correct propagator selection is most consequential. On WGC and GA, PAR1 times are competitive with the best alternatives, confirming that the cascade's confidence-rejection mechanism successfully concentrates the Oracle's effort where it matters most, without penalising performance on easier instances.

5. Related Work

The problem of selecting the most suitable solver or configuration for a given instance has been extensively studied within the broader context of algorithm selection [20] and portfolio solving. The potential benefits of parallel algorithm portfolios were first identified by Huberman et al. [21], who drew an analogy with financial portfolio theory: running multiple algorithms concurrently on the same instance until one succeeds. Gomes and Selman [22] further investigated conditions under which such portfolios outperform their component solvers on constraint programming problems.

In the SAT setting, algorithm selection has seen substantial progress over the past decade. SATzilla [23] predicts algorithm performance using ridge regression and later a pairwise voting scheme based on

random forests. ISAC [24] clusters instances in feature space and applies nearest-neighbour selection on cluster centres. 3S [25] uses k-NN in feature space and introduces pre-solving schedules computed via Integer Linear Programming. SNAPP [26] selects algorithms based on the similarity of predicted performances. While each of these systems is highly effective, they are specialized to a single approach and do not provide a uniform framework across different inputs and performance metrics. A broader overview of runtime prediction techniques at the base of many of these selectors is given in [27], and a systematic comparison of machine learning algorithms for algorithm selection is presented in [28], which led to the development of LLAMA [29], a flexible framework for training and assessing algorithm selection techniques.

Beyond SAT, algorithm selectors have been developed for a range of problems. AQME [30] is a multi-engine selector for QSAT, later extended with self-adaptation of learned selection policies [31]. Stone Soup [32] applies greedy hill climbing to construct algorithm schedules for planning. aspeed [33] computes timeout-minimal algorithm schedules by leveraging ASP's modelling and solving capabilities.

In the ASP setting, claspfolio [34, 35] selects the most promising internal configuration of CLASP using both static and dynamic features, relying primarily on regression-based models. *ME-ASP* [7] is the most directly related prior system: it applies classification-based machine learning to select, on a per-instance basis, the best solver among a set of heterogeneous ASP solvers, exploiting more than 50 cheap-to-compute syntactic features of ground programs, and has been shown to outperform any single solver in the third ASP Competition [36]. Our work extends *ME-ASP* in two ways: we augment its syntactic features with AMOSUM-specific ones, and we adopt a clustering step in the first classification layer inspired by ISAC [24]. As in *ME-ASP*, we treat the propagator as a black box and rely exclusively on static features, ensuring portability to any future system implementing AMOSUM-style constructs without requiring access to solver internals. Calimeri et al. [37] implemented a machine learning technique that selects the best solver according to some specific features of the instantiation produced by the grounder I-DLV.

On the propagation side, dedicated aggregate propagators for SUM constraints have been studied extensively [11, 12]. The AMOSUM construct [6] specialises this line of work by integrating SUM with AMO conditions, which arise naturally in scheduling and resource allocation, and its dedicated propagator significantly outperforms approaches that handle SUM and AMO constraints independently. However, as our experiments confirm, no single propagator configuration dominates across all instances, motivating the adaptive selection strategy proposed in this work.

6. Conclusion

In this work, we proposed a machine learning-based approach for the automatic selection of AMOSUM propagator configurations in ASP. Starting from the observation that no single configuration dominates across all problem instances, we framed the selection task as a classification problem and developed a pipeline that extracts semantically meaningful features capturing the interaction between SUM and AMO constraints. We constructed a diverse training dataset across three combinatorial benchmarks and trained a series of classifiers, including cascade architectures that balance inference speed with predictive accuracy. Experimental results confirm that the proposed selector outperforms any static configuration in terms of both solved instances and average runtime.

As future work, we plan to extend the training dataset to a broader and more heterogeneous set of combinatorial domains. Then, our approach relies exclusively on static features; a natural extension would be to incorporate dynamic features obtained by running the propagator for a short allotted time and inspecting solver-internal statistics, moving towards a portfolio-based strategy in the spirit of claspfolio. This would allow the selector to adapt to instance-specific runtime behaviour and potentially yield further performance gains.

Finally, our implementation and all problem instances used in our experiments are available at https://github.com/instafiore/Amosum-ML/tree/aspocp.

Acknowledgments

This work was supported by the Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by the Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); under PN RIC project ASVIN "Assistente Virtuale Intelligente di Negozio" (CUP B29J24000200005); by Regione Calabria under project IOS4OS "Intelligent Optimization System for Optimization Solutions" (CUP J89I24002920005); under project NextGenGuides "Innovazione Tecnologica per le Guide Turistiche del Futuro tramite l'ausilio di tecnologie innovative AI e sistemi di geolocalizzazione avanzata" (CUP J39I24002040005); under project MOZART "Modelli e tecniche di mOdernizzazione di applicaZioni e data silos a supporto di clienti esterni, field force e impiegati di backoffice basati su AI GeneRativa e apprendimento auTomatico" (CUP J49I24001740005); by the LAIA lab (part of the SILA labs). Mario Alviano is a member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT, and Claude in order to: Grammar and spelling check. After using these tools/services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] V. W. Marek, M. Truszczynski, Stable models and an alternative logic programming paradigm, in: The Logic Programming Paradigm - A 25-Year Perspective, Artif. Intell., Springer, 1999, pp. 375–398. doi:10.1007/978-3-642-60085-2_17.

[2] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, Ann. Math. Artif. Intell. 25 (1999) 241–273. doi:10.1023/A:1018930122475.

[3] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, AI Mag. 37 (2016) 53–68. doi:10.1609/AIMAG.V37I3.2678.

[4] A. A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, E. C. Teppan, Industrial applications of answer set programming, Künstliche Intell. 32 (2018) 165–176. doi:10.1007/S13218-018-0548-6.

[5] P. Ferraris, Logic programs with propositional connectives and aggregates, ACM Trans. Comput. Log. 12 (2011) 25. doi:10.1145/1970398.1970401.

[6] M. Alviano, C. Dodaro, S. Fiorentino, M. Maratea, Amo-aware aggregates in answer set programming, in: Proc. of IJCAI, ijcai.org, 2024, pp. 3215–3223.

[7] M. Maratea, L. Pulina, F. Ricca, The multi-engine ASP solver me-asp, in: Proc. of JELIA, LNCS, Springer, 2012, pp. 484–487. doi:10.1007/978-3-642-33353-8_39.

[8] W. Faber, G. Pfeifer, N. Leone, Semantics and complexity of recursive aggregates in answer set programming, Artif. Intell. 175 (2011) 278–298. doi:10.1016/J.ARTINT.2010.04.002.

[9] M. Gebser, B. Kaufmann, T. Schaub, Conflict-driven answer set solving: From theory to practice, Artif. Intell. 187 (2012) 52–89. doi:10.1016/J.ARTINT.2012.04.001.

[10] C. Dodaro, Design and implementation of modern CDCL ASP solvers, Intelligenza Artificiale 18 (2024) 239–259. doi:10.3233/IA-240019.

[11] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, On the implementation of weight constraint rules in conflict-driven ASP solvers, in: Proc. of ICLP, LNCS, Springer, 2009, pp. 250–264. doi:10.1007/978-3-642-02846-5_23.

[12] W. Faber, N. Leone, M. Maratea, F. Ricca, Look-back techniques for ASP programs with aggregates, Fundam. Informaticae 107 (2011) 379–413. doi:10.3233/FI-2011-408.

[13] T. M. Cover, P. E. Hart, Nearest neighbor pattern classification, IEEE Trans. Inf. Theory 13 (1967) 21–27. doi:10.1109/TIT.1967.1053964.

[14] H. Wang, G. Gupta, FOLD-R++: A scalable toolset for automated inductive learning of default theories from mixed data, in: Proc. of FLOPS, LNCS, Springer, 2022, pp. 224–242. doi:10.1007/978-3-030-99461-7_13.
[15] L. Breiman, J. H. Friedman, R. A. Olshen, C. J. Stone, Classification and Regression Trees, Wadsworth, 1984.
[16] L. Breiman, Random forests, Machine Learning 45 (2001) 5–32. doi:10.1023/A:1010933404324.
[17] T. Chen, C. Guestrin, Xgboost: A scalable tree boosting system, in: Proc. of ACM SIGKDD, ACM, 2016, pp. 785–794. doi:10.1145/2939672.2939785.
[18] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T. Liu, Lightgbm: A highly efficient gradient boosting decision tree, in: Proc. of NeurIPS, 2017, pp. 3146–3154.
[19] L. Rokach, O. Maimon, Top-down induction of decision trees classifiers - a survey, IEEE Trans. Syst. Man Cybern. Part C 35 (2005) 476–487. doi:10.1109/TSMCC.2004.843247.
[20] J. R. Rice, The algorithm selection problem, Adv. Comput. 15 (1976) 65–118. doi:10.1016/S0065-2458(08)60520-3.
[21] B. Huberman, R. Lukose, T. Hogg, An economics approach to hard computational problems, Science (New York, N.Y.) 275 (1997) 51–4. doi:10.1126/science.275.5296.51.
[22] C. P. Gomes, B. Selman, Algorithm portfolios, Artif. Intell. 126 (2001) 43–62. doi:10.1016/S0004-3702(00)00081-3.
[23] L. Xu, F. Hutter, H. H. Hoos, K. Leyton-Brown, Satzilla: Portfolio-based algorithm selection for SAT, J. Artif. Intell. Res. 32 (2008) 565–606. doi:10.1613/JAIR.2490.
[24] S. Kadioglu, Y. Malitsky, M. Sellmann, K. Tierney, ISAC - instance-specific algorithm configuration, in: Proc. of ECAI, FAIA, IOS Press, 2010, pp. 751–756. doi:10.3233/978-1-60750-606-5-751.
[25] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann, Algorithm selection and scheduling, in: Proc. of CP, LNCS, Springer, 2011, pp. 454–469. doi:10.1007/978-3-642-23786-7_35.
[26] M. Collautti, Y. Malitsky, D. Mehta, B. O'Sullivan, SNNAP: solver-based nearest neighbor for algorithm portfolios, in: Proc. of ECML PKDD, LNCS, Springer, 2013, pp. 435–450. doi:10.1007/978-3-642-40994-3_28.
[27] F. Hutter, L. Xu, H. H. Hoos, K. Leyton-Brown, Algorithm runtime prediction: Methods & evaluation, Artif. Intell. 206 (2014) 79–111. doi:10.1016/J.ARTINT.2013.10.003.
[28] L. Kotthoff, I. P. Gent, I. Miguel, An evaluation of machine learning in algorithm selection for search problems, AI Commun. 25 (2012) 257–270. doi:10.3233/AIC-2012-0533.
[29] L. Kotthoff, LLAMA: leveraging learning to automatically manage algorithms, CoRR abs/1306.1031 (2013). arXiv:1306.1031.
[30] L. Pulina, A. Tacchella, A multi-engine solver for quantified boolean formulas, in: Proc. of CP, LNCS, Springer, 2007, pp. 574–589. doi:10.1007/978-3-540-74970-7_41.
[31] L. Pulina, A. Tacchella, A self-adaptive multi-engine solver for quantified boolean formulas, Constraints An Int. J. 14 (2009) 80–116. doi:10.1007/S10601-008-9051-2.
[32] J. Seipp, M. Braun, J. Garimort, M. Helmert, Learning portfolios of automatically tuned planners, in: Proc. of ICAPS, AAAI, 2012.
[33] H. H. Hoos, R. Kaminski, M. Lindauer, T. Schaub, aspeed: Solver scheduling via answer set programming, Theory Pract. Log. Program. 15 (2015) 117–142. doi:10.1017/S1471068414000015.
[34] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. T. Schneider, S. Ziller, A portfolio solver for answer set programming: Preliminary report, in: Proc. of LPNMR, LNCS, Springer, 2011, pp. 352–357. doi:10.1007/978-3-642-20895-9_40.
[35] H. H. Hoos, M. Lindauer, T. Schaub, claspfolio 2: Advances in algorithm selection for answer set programming, Theory Pract. Log. Program. 14 (2014) 569–585. doi:10.1017/S1471068414000210.
[36] F. Calimeri, G. Ianni, F. Ricca, The third open answer set programming competition, Theory Pract. Log. Program. 14 (2014) 117–135. doi:10.1017/S1471068412000105.
[37] F. Calimeri, C. Dodaro, D. Fuscà, S. Perri, J. Zangari, Efficiently coupling the I-DLV grounder with ASP solvers, Theory Pract. Log. Program. 20 (2020) 205–224. doi:10.1017/S1471068418000546.