Synthesizing Feature Extractors of Algorithm Selection for Constraint Optimization

Anonymous author
Anonymous affiliation

Abstract

Algorithm selection for constraint optimization requires extracting features that capture problem structure. Manually engineering feature extractors demands deep domain expertise and becomes a bottleneck whenever a new problem class arises, which is the very reason algorithm selection has been confined to a handful of well-studied domains. This paper presents a problem-specific automated approach that uses Large Language Model (LLM) agents to synthesize executable Python scripts as interpretable feature extractors. Given a MiniZinc model and instance data, an LLM agent operating in a structured insert–check/fix–verify loop produces a Python script that constructs a tailored graph representation of the instance and computes 50 standardized structural and semantic features (e.g., graph density, depot centrality, demand statistics, constraint tightness). We evaluate the generated extractors on a five-solver portfolio (Gurobi, CPLEX, SCIP, Gecode, OR-Tools) for three representative problems: vehicle routing (VRP), car sequencing (CS), and fixed-length error-correcting codes (FLECC). The synthesized features are more diverse (48.5% lower average correlation on VRP), more efficiently utilized (96% effective utilization vs. 57% for hand-crafted features), and yield consistently better selectors across Random Forest, AutoFolio, LLAMA, and Auto-sklearn pipelines. Compared to the expert-curated *mzn2feat* extractor, our *LLM2feat* extractors deliver absolute test-accuracy improvements of up to +5.2 percentage points on FLECC, +2.7 on VRP, and +2.8 on CS, and outperform every released transformer-based *trans2feat* variant on the two problems for which features are public, all while remaining fully interpretable.

2012 ACM Subject Classification Computing methodologies → Machine learning

Keywords and phrases Algorithm selection, constraint optimization, feature extraction, large language models, MiniZinc, program synthesis

Digital Object Identifier 10.4230/LIPIcs...

Category Doctoral Program

1 Introduction

Combinatorial optimization underlies many real-world tasks, from vehicle routing and scheduling to error-correcting code design. Modern solver portfolios spanning constraint programming (CP), mixed-integer programming (MIP), and hybrid systems offer complementary strengths, but no single solver dominates across instances of even one problem class. The Algorithm Selection (AS) problem [13, 5] addresses this by predicting, before solving, which solver from a portfolio $\mathcal{P}$ will perform best on a given instance.

A key bottleneck of AS, however, is the need for an effective *feature extractor*: a function Φ mapping an instance i to a vector $\phi(i) \in \mathbb{R}^d$ that captures structural and semantic properties relevant to solver behaviour. Designing such an extractor requires deep domain expertise and intimate knowledge of which features correlate with solver performance, and it must be cheap enough that extraction does not dominate solver runtime. As a consequence, the small set of problem domains where AS has been deployed successfully is largely determined by the availability of high-quality extractors, leaving most problem classes outside the reach of portfolio techniques.

A common workaround is to translate a high-level model into a flat representation (e.g., FlatZinc) and reuse generic extractors [1, 2]. This loses precisely the structural

information that distinguishes hard from easy instances: global constraints become long lists of primitives, and the incidence structure of variables and constraints is obscured. Recent neural approaches [8, 16, 11] learn dense embeddings end-to-end, but at the cost of interpretability: individual dimensions carry no clear semantic meaning, cannot be inspected by domain experts, and resist being refined by hand.

Approach.

This paper investigates a problem-specific framework that automates the construction of feature extractors using LLM agents while preserving interpretability. Given the MiniZinc model file (`.mzn`), one or more instance data files (`.dzn`), and a small data schema, an LLM agent operating inside a structured insert–check/fix–verify loop synthesizes an executable Python script. That script is the feature extractor: it parses each instance, builds a tailored graph representation, and computes a fixed-size, interpretable feature vector (50 features in our experiments). Because the artefact is human-readable Python rather than a learned neural network, the resulting extractor can be audited, validated, and refined — a "gray box" that complements rather than replaces domain expertise.

This research forms the problem-specific component of a broader automated feature engineering for combinatorial optimization. The companion problem-generic line of work, which targets arbitrary MiniZinc problems through a universal bipartite-graph representation, is outside the scope of this paper; here we focus exclusively on the problem-specific setting where sufficient instances of a single problem are available for training.

Contributions.

1. A problem-specific LLM-agent framework for synthesizing interpretable Python feature extractors from a MiniZinc model, coupled with a structured prompt design (script-system and `mzn`-tuning) that drives the insert–check/fix–verify loop.
2. An empirical study on three constraint optimization problems — vehicle routing (VRP) [12], car sequencing (CS) [11], and fixed-length error-correcting codes (FLECC) [11] — using a five-solver portfolio (Gurobi, CPLEX, SCIP, Gecode, OR-Tools) and four AS pipelines (Random Forest, AutoFolio, LLAMA, Auto-sklearn).
3. Quantitative evidence that the synthesized features are more diverse, more efficiently utilized, and ultimately more predictive than the established *mzn2feat* [2] and the transformer-based *trans2feat* [11] extractors, with up to +5.2 percentage points absolute accuracy improvement and consistently better average solver ranking across all three problems.

2 Related Work

Algorithm Selection and Feature Extractors.

Since Rice's seminal formulation [13], AS has matured into a cornerstone technique for solving combinatorial problems [5]. For MiniZinc, *mzn2feat* [1, 2] is the de-facto standard: it produces 95 features by analysing the flattened FlatZinc form of an instance. Standard AS toolchains, including AutoFolio [7], LLAMA [6], and Auto-sklearn [3], consume such feature vectors as input. Our framework is orthogonal to the AS toolchain: it produces drop-in feature vectors that can be fed to any existing selector.

Neural Embeddings.

Loreggia et al. [8] use convolutional networks on rendered text representations; Zhang et al. [16] combine graph neural networks with expert features for SAT solver selection; Pellegrino et al. [11] train transformers on FlatZinc to obtain 116-dimensional embeddings (*trans2feat*) for AS. These approaches automate feature extraction but yield embeddings whose dimensions lack semantic meaning. We compare directly against *trans2feat* on the two problems for which the authors released features (CS and FLECC) and show that our interpretable extractors outperform every released variant.

LLMs for Feature Engineering.

CAAFE [4] pioneers LLM-driven feature synthesis for tabular ML datasets, generating Python transformations of existing columns. The closest in spirit to our work is the recent MCP-based agentic framework of Szeider [15], which connects LLMs with symbolic CSP solvers. We diverge from both by targeting the *declarative problem specification* (a MiniZinc `.mzn` model and `.dzn` data), generating a self-contained executable extractor, and producing structural features anchored in a tailored graph representation rather than transformations of pre-existing features.

3 Problem-Specific Framework

3.1 Preliminaries

The Algorithm Selection problem [13, 5] considers a portfolio $\mathcal{P}$ of algorithms, a set of instances I, a performance metric $PM(A, i)$, and a budget B. Each instance is described by a feature vector $\phi(i) \in \mathbb{R}^d$ produced by an extractor $\Phi : i \mapsto \phi(i)$. The AS task is to learn a selector $S : \mathbb{R}^d \to \mathcal{P}$ that maximizes $\sum_{i \in I} PM(S(\phi(i)), i)$ subject to B. MiniZinc [10, 9, 14] is a high-level declarative modelling language; a MiniZinc model (`.mzn`) declares variables, constraints, and an optional objective, and is paired with a data file (`.dzn`) that supplies parameters. We define the *Single Best* solver (SB) as the portfolio member with the highest aggregate PM over I, and the *Virtual Best* solver (VBS) as the per-instance oracle.

We use an LLM agent of the form $\mathcal{A} = (L, T, M, \pi, E)$, where L is the language model, T is a set of tools, M a memory, π a prompting policy, and E the environment. The agent operates in a loop $o_t \xrightarrow{\pi} p_t \xrightarrow{L} a_t \xrightarrow{T,E} o_{t+1}$. In our setting, the tools manipulate a Python script (clear, insert, check, fix, execute), the memory holds intermediate output, and the environment is a sandbox that runs the script and captures stdout/stderr.

3.2 Synthesis Procedure

The agent is configured by two prompt files. The `script_system` prompt establishes general behavioural rules: produce a single self-contained Python file, follow a strict workflow, use only allowed tools, and emit standardized output via a helper function. The `mzn-tuning` prompt is domain-specific: it instructs the agent to read the MiniZinc model and data, build a tailored graph representation, extract exactly 50 features (problem-size, graph properties, distributional statistics, structural complexity), and return them in a fixed schema. Template substitution replaces `${INSTANCE}`, `${SCHEMA}`, `${MODEL}`, and `${PROBLEM}` with the relevant content. The agent then iterates the four-step loop in Figure 1:

1. **Clear**: empty the script buffer.
2. **Insert**: write a complete Python script.

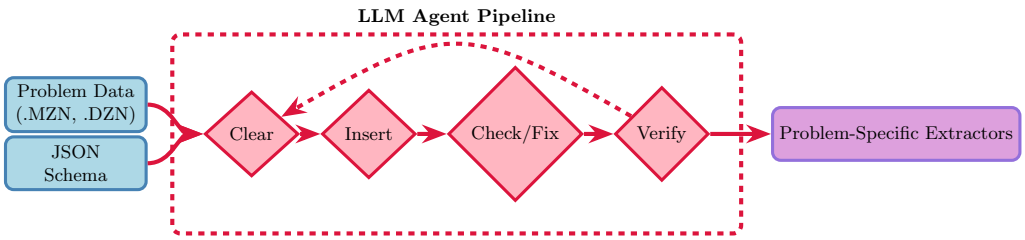


Figure 1 Workflow of the problem-specific feature-extractor synthesis. The LLM agent receives the MiniZinc model (`.mzn`), one instance (`.dzn`), and a schema, and runs an insert–check/fix–verify loop until an executable Python extractor is produced.

```python
from lmtune_helpers import input_data, output_results
import networkx as nx
import numpy as np

def main():
    instance_data = input_data()  # parsed from the .dzn file
    # Build a tailored graph: nodes encode variables and constraints
    # specific to the problem (e.g., customers and routes for VRP).
    G = build_problem_graph(instance_data)
    results = {
        "README": "...methodology description...",
        "characteristic_1": ...,   # problem-size metric
        # ... 48 more features (graph properties, statistics,
        # symmetry, tightness, distribution skewness, ...)
        "characteristic_50": ...,  # structural complexity
    }
    output_results(results)
```

Listing 1 Schematic of the synthesized Python extractor (problem-specific template).

3. **Check/Fix**: validate syntax and prompt requirements; repair any issues.
4. **Execute**: run the script on the supplied instance, capture and validate the output dictionary.

The loop terminates as soon as the script produces a valid 50-feature dictionary. We empirically observed that all four steps are necessary: removing any of them causes the agent to fail to produce an executable extractor (Section 4).

The output is a single Python script that takes any instance of the same problem as input and emits a 50-dimensional feature vector. The script is human-readable, can be inspected and edited by domain experts, and runs on CPU within seconds per instance.

4 Experiments

4.1 Setup

Solvers.

We use five solvers from the MiniZinc Challenge[1]: Gurobi 12.0.3, CPLEX 22.1.2, SCIP 9.2.3, Gecode 6.2.0, and OR-Tools 9.3.10497. The portfolio spans commercial MIP (Gurobi,

[1] `https://www.minizinc.org/challenge/2025/results/`

CPLEX), hybrid CP/MIP (SCIP), pure CP (Gecode), and the recent gold-medal-winning OR-Tools.

Problems.

We evaluate on three problem classes for which sufficient instances are available: vehicle routing (VRP) [12], car sequencing (CS) [11], and fixed-length error-correcting codes (FLECC) [11]. Instances are split 70 %/30 % into training and test sets; the same split is reused across all extractors and AS toolchains for fairness.

Solving and extraction protocol.

We run each solver on each instance with a 20-minute timeout, following MiniZinc Challenge conventions. The performance metric $PM(A, i)$ is the objective value at timeout for optimization problems and the wall-clock runtime for decision problems. The extractor-generation loop is bounded by a 1-minute LLM wall-clock budget; we use OpenAI `o4-mini-2025-04-16` as L in this study. All solver runs use a compute cluster with two AMD 7403 processors (24 cores at 2.8 GHz, 32 GB RAM/core).

AS toolchains.

We feed each feature set to four AS pipelines: Random Forest (RF) with 300 estimators, AutoFolio (AF), LLAMA, and Auto-sklearn (AutoSK) [3]. RF and AutoSK are trained both with the accuracy (*Acc*) loss and the average-ranking (*Rank*) loss; AF and LLAMA use their default settings. We report two metrics: *Acc*, the fraction of instances on which the selector picks the actual best solver, and *Rank*, the average rank position of the selected solver (lower is better).

Research questions.

Q1 How diverse are the features synthesized by *LLM2feat*, compared to *mzn2feat*?
Q2 How efficiently does each feature contribute to the trained selectors?
Q3 How does AS accuracy compare across *LLM2feat*, *mzn2feat*, and the transformer-based *trans2feat*?

4.2 Q1: Feature Correlation Analysis

For each feature set we trained an RF selector and extracted the top 20 features by importance, then computed pairwise Pearson correlations. Figure 2 shows the resulting heatmap for VRP: *LLM2feat* features are visibly less correlated, with average $|r| = 0.221$ versus 0.429 for *mzn2feat* (a 48.5 % reduction). The same trend holds for FLECC ($|r| = 0.306$ vs. 0.330) and for CS ($|r| = 0.551$ vs. 0.725, a 24 % reduction). Lower correlation indicates that the LLM-synthesized features capture more complementary aspects of problem structure, which in turn supports more efficient selectors.

4.3 Q2: Feature Utilization Efficiency

We measure utilization by computing RF feature importances and counting features above an importance threshold of 0.001 ("effectively utilized"). Figure 3 reports importance distributions for the full feature sets. *LLM2feat* dominates on all three problems: 96 % utilization on VRP (vs. 56.8 % for *mzn2feat*), 58 % on FLECC (vs. 23.2 %), and 84 % on

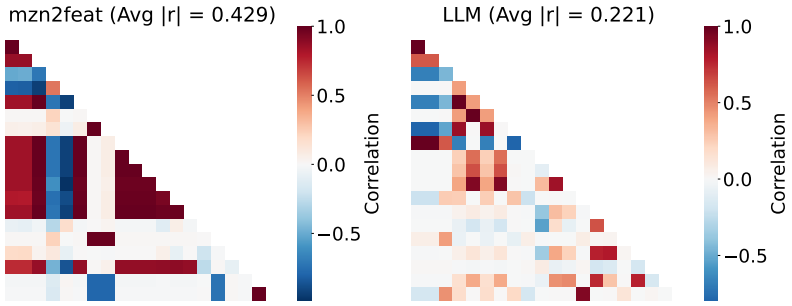
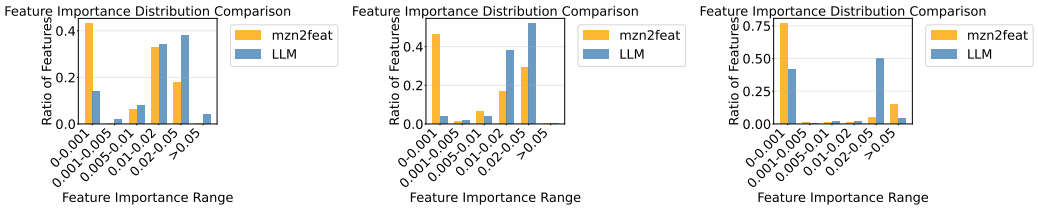


Figure 2 Feature correlation heat-map on VRP (top-20 features by RF importance, axis labels removed for clarity). *LLM2feat* features are 48.5 % less correlated on average ($|r| = 0.221$) than *mzn2feat* features ($|r| = 0.429$).



(a) FLECC **(b)** CS **(c)** VRP

Figure 3 Feature-importance distributions. *LLM2feat* features spread weight across more importance bins; *mzn2feat* concentrates on a few features and leaves many unused.

CS (vs. 45.1 %). On VRP this is a 69 % relative improvement in effective feature usage. The pattern reflects the LLM's tendency to design features along distinct structural and semantic axes, whereas *mzn2feat* contains many overlapping flat-encoding statistics whose contributions are absorbed into a small subset of important dimensions.

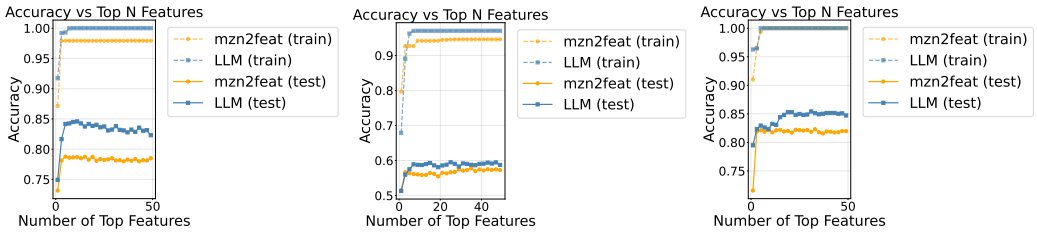
4.4 Q3: Algorithm Selection Accuracy

We first study how accuracy scales with the number of top-k most important features (Figure 4). On all three problems, *LLM2feat* reaches a higher plateau than *mzn2feat*, and continues to improve with additional features past the point where *mzn2feat* stagnates. On VRP, for instance, beyond 10 features *mzn2feat*+RF saturates around 81 % test accuracy; *LLM2feat*+RF reaches its peak only after 20 features.

Table 1 summarizes test-set accuracy and average ranking with each AS toolchain trained under the *Acc* loss. *LLM2feat* dominates on every problem and every selector. The strongest absolute improvement is on FLECC, where *LLM2feat*+AutoSK reaches 84.6 % accuracy compared to 79.4 % for *mzn2feat*+AutoSK and 78.4 % for the best transformer variant. On CS the improvement is +1.9 pp over *mzn2feat* and +9.5 pp over *trans2feat*. AutoFolio reduces to the Single-Best baseline on all three problems regardless of the feature set, since its internal model selection collapses to a constant predictor on these data.

Comparison with transformer features.

For CS and FLECC we retrained selectors with each of the 20 *trans2feat* variants released by Pellegrino et al. [11] (alternating "com-k" and "neural-k" configurations). On CS, the best *trans2feat*+AutoSK reaches 54.5 % accuracy; even the worst-performing of our 10 independent



(a) FLECC **(b)** CS **(c)** VRP

Figure 4 Test accuracy as a function of the number of top-k important features. *LLM2feat* reaches a higher plateau and benefits from more features than *mzn2feat*.

Table 1 Test-set accuracy (*Acc*) and average ranking (*Rank*) on VRP, CS, and FLECC under the *Acc* training loss. SB = Single-Best baseline. *trans2feat* features were released only for CS and FLECC. Bold marks the best per column.

	VRP		CS		FLECC	
	Acc	*Rank*	*Acc*	*Rank*	*Acc*	*Rank*
SB	79.5%	1.221	49.0%	**1.616**	78.2%	1.420
mzn2feat+AF	79.5%	1.221	49.0%	**1.616**	78.2%	1.420
mzn2feat+RF	83.0%	1.184	58.5%	1.680	79.5%	1.391
mzn2feat+LLAMA	82.9%	1.184	59.4%	1.798	78.7%	1.397
mzn2feat+AutoSK	84.4%	1.166	62.1%	1.787	79.4%	1.394
trans2feat+RF	—	—	52.9%	2.004	77.3%	1.447
trans2feat+AutoSK	—	—	54.5%	1.931	78.4%	1.417
LLM2feat+AF	79.5%	1.221	49.0%	**1.616**	78.2%	1.420
LLM2feat+RF	85.7%	**1.155**	61.3%	1.681	83.5%	1.338
LLM2feat+LLAMA	**85.8%**	1.157	61.0%	1.773	84.2%	**1.306**
LLM2feat+AutoSK	85.4%	1.160	**64.0%**	1.738	**84.6%**	1.310

LLM2feat runs achieves 52.8 % on the same selector and the typical *LLM2feat* run reaches 63 %–64 %. On FLECC, the best *trans2feat*+RF run yields 77.3 % test accuracy, while every *LLM2feat* run we generated exceeds 81 %. The advantage holds across both RF and AutoSK and despite *trans2feat* using more than twice as many features (116 vs. 50).

4.5 Qualitative Inspection of Synthesized Features

A practical benefit of producing extractors as Python code is that the generated features are explicit and can be read by a human reviewer. On VRP, the synthesized extractor introduces features such as `avg_demand` (mean customer demand), `depot_centrality` (closeness centrality of the depot in the customer graph), and route-length lower bounds derived from a minimum spanning tree of customer locations. These quantities have well-understood relationships to solver performance: high depot centrality and uniform demand tend to favour MIP solvers (Gurobi, CPLEX, SCIP), whereas heterogeneous demand and clustered customers favour CP-style search (Gecode, OR-Tools). The LLM rediscovers these problem-specific cues without being told to look for them, and the resulting extractor remains a small, auditable Python program. Similar problem-tailored features appear in the CS and FLECC extractors (e.g., option compatibility densities, code-distance distributions), which we omit here for space.

Stability of synthesis.

We ran the synthesis loop 10 times per problem (different LLM samples, identical prompts). The resulting selectors fluctuate by less than 2 percentage points in *Acc* and less than 0.04 in *Rank* across runs, and every run outperforms the corresponding *mzn2feat*+selector pair. Failure modes were rare: at most one of 10 runs per problem timed out under our 1-minute budget, and these were excluded as the synthesis pipeline allows automatic restarts.

Cost.

A single extractor synthesis on `o4-mini` averaged 17.3 agent iterations, 210 seconds wall-clock, and $0.27 in API cost (≈260k input tokens, 19k output tokens). The cost is paid once per problem class; the generated extractor can then process arbitrarily many instances at no additional LLM cost, making the up-front budget negligible against the 20-minute solver runs it informs.

5 Conclusion and Future Work

We presented a problem-specific framework that uses LLM agents to automatically synthesize interpretable Python feature extractors for algorithm selection on MiniZinc problems. On VRP, CS, and FLECC, the synthesized extractors yield more diverse and more efficiently utilized feature sets than expert-curated *mzn2feat* features and outperform transformer-based *trans2feat* embeddings, while remaining fully auditable Python code that domain experts can inspect, validate, and refine.

This work is part of a broader doctoral programme on automated feature engineering for combinatorial optimization. Three immediate research directions follow. First, we plan a controlled study of prompt-design choices (schema granularity, graph-builder design, number of extracted features) to understand which agent-level decisions drive the observed accuracy gains. Second, we are investigating whether the synthesized extractors transfer across structurally similar problems (e.g., from VRP to other routing variants) without re-running the LLM agent. Third, we will combine the present extractors with portfolio-aware loss functions (e.g., MiniZinc Challenge Borda scoring) to optimize selectors directly for the target competition metric. Together with a companion problem-generic line of work, the long-term goal is to make algorithm selection routinely available for new constraint optimization domains.

References

1 Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Features for building CSP portfolio solvers. *CoRR*, abs/1308.0227, 2013. URL: `http://arxiv.org/abs/1308.0227`.

2 Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An enhanced features extractor for a portfolio of constraint solvers. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pages 1357–1359. ACM, 2014. `doi:10.1145/2554850.2555114`.

3 Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Auto-sklearn 2.0: Hands-free automl via meta-learning. *arXiv:2007.04074 [cs.LG]*, 2020.

4 Noah Hollmann, Samuel Müller, and Frank Hutter. Large language models for automated data science: Introducing CAAFE for context-aware automated feature engineering. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

5 Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evol. Comput.*, 27(1):3–45, 2019. doi:10.1162/EVCO_A_00242.

6 Lars Kotthoff. LLAMA: leveraging learning to automatically manage algorithms. Technical Report arXiv:1306.1031, June 2013. URL: http://arxiv.org/abs/1306.1031.

7 M. Lindauer, H. Hoos, F. Hutter, and T. Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015.

8 Andrea Loreggia, Yuri Malitsky, Horst Samulowitz, and Vijay A. Saraswat. Deep learning for algorithm portfolios. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 1280–1286. AAAI Press, 2016. doi:10.1609/AAAI.V30I1.10170.

9 Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the zinc modelling language. *Constraints An Int. J.*, 13(3):229–267, 2008. doi:10.1007/S10601-008-9041-4.

10 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming - CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

11 Alessio Pellegrino, Özgür Akgün, Nguyen Dang, Zeynep Kiziltan, and Ian Miguel. Transformer-Based Feature Learning for Algorithm Selection in Combinatorial Optimisation. In *31st International Conference on Principles and Practice of Constraint Programming (CP 2025)*, volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.CP.2025.31.

12 Eduardo Queiroga, Ruslan Sadykov, Eduardo Uchoa, and Thibaut Vidal. 10,000 optimal CVRP solutions for testing machine learning based heuristics. In *AAAI-22 workshop on machine learning for operations research (ML4OR)*, 2021.

13 John R. Rice. The algorithm selection problem. volume 15 of *Advances in Computers*, pages 65–118. Elsevier, 1976. doi:10.1016/S0065-2458(08)60520-3.

14 Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The minizinc challenge 2008–2013. *AI Magazine*, 35(2):55–60, 2014. doi:10.1609/aimag.v35i2.2539.

15 Stefan Szeider. Bridging language models and symbolic solvers via the model context protocol. In *28th International Conference on Theory and Applications of Satisfiability Testing, SAT 2025, August 12-15, 2025, Glasgow, Scotland*, volume 341 of *LIPIcs*, pages 30:1–30:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.SAT.2025.30.

16 Zhanguang Zhang, Didier Chételat, Joseph Cotnareanu, Amur Ghose, Wenyi Xiao, Hui-Ling Zhen, Yingxue Zhang, Jianye Hao, Mark Coates, and Mingxuan Yuan. Grass: Combining graph neural networks with expert knowledge for SAT solver selection. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*, pages 6301–6311. ACM, 2024. doi:10.1145/3637528.3671627.