Learning to Solve and Optimize by Evolving Code

Veronika Semmelrock[1] , **Benedetta Strizzolo**[2] , **Francesco Zuccato**[1] , **Gerhard Friedrich**[1] , **Patrick Rodler**[1] and **Konstantin Schekotihin**[1]
[1]University of Klagenfurt, Austria
[2]University of Udine, Italy
{firstname.lastname}@aau.at, strizzolo.benedetta@spes.uniud.it

Abstract

Combinatorial and optimization problems are fundamental to many industrial AI applications. Solving large-scale real-world instances of such problems typically requires careful problem formalization, specialized solvers, and expert-designed heuristics. Thus, experts need to specify not only *what* solutions are, but also *how* they are derived.

By introducing the tool CHECKMATE, we show that algorithm generation via code evolution represents a paradigm shift by eliminating the need to formulate the *how*. CHECKMATE solely relies on the *what*. Specifically, a formal specification ensures solutions' correctness and enables systematic performance evaluation of the generated programs, while a natural language description guides the evolutionary process.

The effectiveness of our method is demonstrated on selected problems from two industrial domains: configuration and scheduling. In all cases, the evolved algorithms consistently outperform state-of-the-art solvers. This underscores the potential of formal methods in guiding code evolution for automatically solving complex real-world problems.

1 Introduction

Combinatorial and optimization problems are central to many industrial AI applications, including the automated engineering of technical systems (e.g., configuring and planning the composition of large electronic systems) [Falkner *et al.*, 2016] and the automation of process planning and scheduling [Da Col and Teppan, 2022]. A long-standing vision in AI is that domain experts should specify *what* constitutes a correct solution, while the computer should determine *how* to obtain it efficiently [Freuder, 2018].

In practice, however, applying state-of-the-art solvers from constraint [Laborie *et al.*, 2018], logic [Kaufmann *et al.*, 2016], or mathematical programming [Perron *et al.*, 2023] to large real-world problem instances still demands substantial expert intervention beyond the *what*. Effective deployment often requires redesigning problem specifications [Dodaro *et al.*, 2024], crafting problem-specific heuristics [Comploi-Taupe *et al.*, 2023], decomposing the problem into manageable subproblems [El-Kholany *et al.*, 2025], or implementing tailored local-search procedures [Sanghikian *et al.*, 2026].

To address these challenges, we build on recent advances in program generation via code evolution [Novikov *et al.*, 2025]. Our approach integrates a new CHECKMATE component into the OpenEvolve [Sharma, 2025] controller loop. Given a natural-language description and a formal specification of *what* solutions are, along with a training set of representative instances, OpenEvolve+CHECKMATE automatically synthesizes a problem-specific solver—implemented as a Python program—without requiring any formulation of *how* to solve the problem.

We investigate the following research questions:

- *RQ1* Can OpenEvolve+CHECKMATE solve hard real-world *(a)* combinatorial and *(b)* optimization problems, such as configuration and scheduling?
- *RQ2* How does the performance of the generated programs compare to state-of-the-art solvers?
- *RQ3* How well do these evolved programs scale?

Our evaluation targets configuration problems from automated engineering and an industrial scheduling problem. We analyze two configuration tasks from Siemens that capture pivotal real-world challenges: one stresses solver scalability with respect to solution size [Semmelrock and Friedrich, 2025], and the other examines solver behavior when several difficult problems are combined [Gebser *et al.*, 2015]. In addition, we evaluate a real-world scheduling problem from metalworking at voestalpine [Zuccato *et al.*, 2025].

Across all tasks, OpenEvolve+CHECKMATE generated problem-specific code that outperformed the respective state-of-the-art solvers by orders of magnitude on large or hard instances. Thus, our approach extends to solving problems where leading solvers cannot deliver solutions.

2 Preliminaries

Since our contribution integrates with OpenEvolve [Sharma, 2025], an open-source implementation of AlphaEvolve [Novikov *et al.*, 2025], we provide a brief overview of this framework. OpenEvolve is a code optimization framework that automatically generates programs capable of solving specified problems by leveraging evolutionary computation and the capabilities of Large Language Models (LLMs) in program synthesis. This framework operates through

an asynchronous pipeline consisting of four core modules: *(i)* a *prompt sampler* that generates prompts using previously evolved programs and their evaluation scores, *(ii)* an *LLM ensemble*, i.e., a set of LLMs that process prompts and produce full code rewrites or targeted edits, *(iii)* an *evaluator pool* that scores and ranks generated programs, and *(iv)* a *program database* that stores these programs and their scores in a structured grid [Mouret and Clune, 2015], organizing them according to diverse user-defined criteria, such as code complexity and correctness. Note that the evaluator pool, at its core, is an evaluation function that the user must implement. An island-based genetic algorithm [Tanese, 1989] manages selection, migration, and evolution across iterations.

Adopting OpenEvolve's terminology, we denote an *artifact* as a textual error-feedback provided to the LLM, and the *combined score* $z_j \in [0, 1]$ as a numerical score assigned to program p_j. Intuitively, the higher the combined score, the higher the likelihood for the program to be selected by the prompt sampler and to guide further evolution.

3 Approach

Extending OpenEvolve's pipeline (Sec. 2), our approach CHECKMATE provides a general, problem-independent implementation of the *evaluator pool* that employs state-of-the-art solvers as verifiers to check the correctness of programs' produced problem solutions. Fig. 1 illustrates the integrated framework, with our contributions highlighted in the dashed blue box. OpenEvolve+CHECKMATE expects the following inputs: *(i)* a natural language description D of the problem (prompt), *(ii)* a formal specification F defining valid problem solutions, *(iii)* a solution verifier V, *(iv)* a set of training problem instances I (for which valid problem solutions are assumed to exist), *(v)* a set S of scoring functions for program evaluation, *(vi)* a (possibly empty) initial program p_0, and *(vii)* a set of evolution and LLM hyperparameters Λ.

Given the inputs, OpenEvolve+CHECKMATE, denoted as the function generator g, produces a program p^*. Formally:

$$g : (D, F, V, I, S, p_0, \Lambda) \mapsto p^*$$

The generation of p^* involves N training iterations, with N defined in Λ. At each iteration $1 \leq j \leq N$, the LLM ensemble non-deterministically generates an intermediate program p_j, aiming to improve the programs $p_0, \ldots, p_{j-1}$. CHECKMATE evaluates p_j by producing the combined score z_j. After N iterations, the p_j achieving the highest z_j will then correspond to the final best program p^*.

At the core of CHECKMATE, the solution verifier V is used to check the correctness of program outputs. Specifically, given a program p_j, which (possibly) generates a candidate solution c_{ji} for a problem instance i, V outputs *(i)* a $verdict \in \{correct, incorrect\}$ indicating whether c_{ji} is an (in)correct solution for i, and, *(b)* in case of $incorrect$, some $feedback$ detailing which parts of the formal specification F are violated by c_{ji}. Formally:

$$p_j : i \mapsto c_{ji} \qquad V : (F, i, c_{ji}) \mapsto \langle verdict, feedback \rangle$$

CHECKMATE's overall program evaluation process, employing V, is detailed next.

3.1 Program Execution and Evaluation

In each training iteration $j \leq N$, in order to evaluate the generated program p_j, CHECKMATE:

(1) Executes p_j on a training instance $i \in I$: if p_j fails, go to (4) and forward $\langle incorrect, feedback \rangle$, where the feedback details the failure reason, e.g., an error stack trace; otherwise, go to (2) and forward candidate solution c_{ji};
(2) Syntactically checks c_{ji}: if the check fails, go to (4) and forward $\langle incorrect, feedback \rangle$, where the feedback includes, e.g., syntactic errors in c_{ji}; else, go to (3) and forward c_{ji} parsed into the verifier-specific format;
(3) Checks the correctness of c_{ji} using the verifier V: if the check returns positively, go to (4) and forward $\langle correct, \emptyset \rangle$; else, go to (4) and forward $\langle incorrect, feedback \rangle$, where feedback includes, e.g., violated constraints in F;
(4) Stores $feedback$ and evaluates p_j's performance on instance i using the scoring functions S, based on the verdict ($correct$/$incorrect$) as well as statistics such as runtime, consumed memory, and resulting objective values.

This process is repeated for all training instances $i \in I$, or until the early stopping is triggered due to k consecutive $incorrect$ verdicts.[1]

Finally, CHECKMATE *(i)* aggregates the instance-level scores from Step (4) to determine z_j, the combined score of p_j, and *(ii)* returns z_j with the stored feedback for all instances (i.e., artifacts) to OpenEvolve's main controller loop.

3.2 Textual Feedback

The artifacts (cf. Sec. 2) are used to provide textual feedback directly to the LLM. They depend on the underlying failure type, of which we distinguish five kinds: *(i) execution* exceptions, such as compilation or runtime errors, *(ii) intentional* exceptions, raised by the evolved program itself due to a precondition or invariant violation, *(iii) exceeded resource* errors, whenever (user-defined) time or memory limits are reached, *(iv) syntactic* errors, e.g., an incomplete candidate solution, and *(v) semantic* errors, if the verdict of V for c_{ji} equals $incorrect$.

Each artifact comprises the failed instance, the failure type, and a suggestion for repairing the failure. The suggestion proposes an improvement to the program in natural language, depending on the failure type. It can include the stack trace, the error message, the position of syntactic errors, or the set of violated constraints or logical sentences. The artifact is stored alongside program p_j and is injected into the prompt whenever p_j is selected in subsequent iterations.

Failure Protocol: Self-refined feedback on program-level. To support the evolution process when programs do not produce candidate solutions, we introduce the Failure Protocol as part of the textual feedback mechanism. By instructing the LLM via the prompt to implement this protocol, we enable a program-level self-refinement loop. Whenever the program expects that the ongoing candidate solution generation will fail, it should raise one of the following exceptions, indicating that it believes *(i)* the instance has no correct solution,

[1] On early stopping, unevaluated instances are deemed *incorrect*.

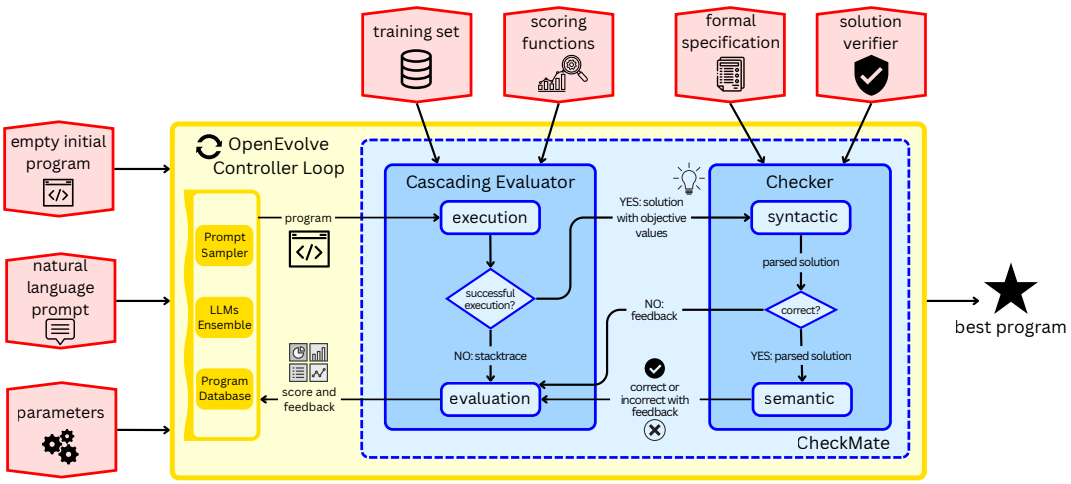

Figure 1: System overview of the proposed evolution framework

(ii) the instance has a correct solution but the implemented search strategy will not find it, or *(iii)* it cannot recover from wrong decisions.

Exceptions raised in accordance with the Failure Protocol are referred to as *intentional* exceptions. They include additional information that the program considers useful feedback for evolution. During execution, CHECKMATE intercepts intentional exceptions and parses them into artifacts, including the suggestion that the error indeed lies within the program, because all instances are assumed to be satisfiable.

3.3 Inputs

In this section, we present the inputs to the code evolution framework that are either novel or central to our approach.

Formal specification and solution verifier. The formal specification F is used by the verifier V to check if a candidate solution c_{ij} yielded by the program p_j is correct for the problem instance i. It can be provided, e.g., in the form of constraints or logical sentences depending on the selected V, such as clingo [Gebser *et al.*, 2019], or CP Optimizer [Laborie *et al.*, 2018]. Since F is used solely to check, there is no need to tune it for solving, e.g., by employing guessing or symmetry-breaking techniques.

Prompt. The natural language description D is a solver-agnostic specification designed to be independent from the format of the solution verifier. It is structured as a domain-specific prompt which contains: *(i)* the LLM identity, *(ii)* the task to be performed, *(iii)* the problem description including context, core entities, and constraints that must be satisfied, and *(iv)* instructions on expected input/output formats and algorithm requirements. This structured approach provides the LLM with the necessary context for generating programs.

Scoring functions. The scoring functions constitute one of the most fundamental components of the learning process, as they affect how programs are ranked, selected, and evolved across iterations. As introduced in Sec. 3.1, they are employed to derive z_j, the combined score of each program p_j. Specifically, z_j is built upon the correctness z_j^c and the quality-efficiency z_j^{qe} trade-off scores. The correctness score builds upon the number of training instances $i \in I$ solved, while the quality-efficiency score combines the quality score z_j^q and the efficiency score z_j^e. The quality score reflects the optimization of the objective values, whereas the efficiency score combines the runtime and memory consumption.

In particular, CHECKMATE *(i)* normalizes the raw statistics into scores in $[0, 1]$;[2] *(ii)* aggregates the instance-level scores into the overall correctness z_j^c, quality z_j^q, and efficiency z_j^e scores; *(iii)* combines the quality and efficiency scores into the composite quality-efficiency tradeoff score z_j^{qe}; and *(iv)* combines z_j^c, z_j^{qe} into the combined score z_j. To address these tasks, we employed an exponential decay function for Step (i), the arithmetic mean for (ii), and combinations of harmonic mean, product, and Prioritized Ordered Weighted Average ($POWA$) [Yager, 2009] for (iii) and (iv).

Training set. The training set should include representative instances of various difficulty levels. Since the initial program p_0 is empty, including sufficiently easy instances is highly recommended to enable the evolutionary process to get started. In early iterations, the main challenge is to solve at least some instances. Once this occurs, the scoring function becomes informative, yielding non-zero correctness values and therefore a useful combined score, enabling the program to evolve effectively. As training progresses, it appears that instances of moderate difficulty promote generalization, while hard ones support scaling and efficiency improvements.

4 Case Studies

In this work, we demonstrate the effectiveness of our approach on the three case studies motivated in Sec. 1. This section provides a high-level overview of these case studies.

[2]Normalization must ensure the maximization of each score. For decision problems, we assign a trivial quality score of 1 to any correct solution, whereas for optimization problems, we require lower and upper bounds for each objective of every instance for scoring.

4.1 House Configuration Problem

The technology-independent House Configuration Problem (HCP) [Fleischanderl *et al.*, 1998; Friedrich *et al.*, 2011] was introduced by Siemens and abstracts electronic modules, frames, and racks into things, cabinets, and rooms. Specifically, persons own multiple things, whereas each thing belongs to exactly one person. The task is to assign things to cabinets and cabinets to rooms while satisfying capacity, ownership, and ordering constraints (the latter introduced in [Semmelrock and Friedrich, 2025]).

4.2 Combined Configuration Problem

The Combined Configuration Problem (CCP) [Gebser *et al.*, 2015] is a combinatorial benchmark motivated by industrial product configuration tasks at Siemens, such as the configuration of railway control and safety systems [Falkner *et al.*, 2016]. The problem is defined via a directed acyclic graph with vertices, paths, bins, colors, areas, and border elements. The CCP integrates multiple interacting subproblems that must be solved simultaneously: *(P1)* vertex coloring, *(P2)* bin packing, *(P3)* partitioning into disjoint paths, *(P4)* matching border elements to areas, and *(P5)* ensuring the connectedness of color-induced subgraphs. The combination of these subproblems makes the CCP very challenging for solvers.

4.3 Energy-Aware Double-Flexible Job-Shop

The Energy-Aware Double-Flexible Job Shop Problem (E-DFJSP) scheduling problem [Gong *et al.*, 2018] exemplifies a real-world production process at voestalpine[3], where jobs consist of multiple metal-cutting operations. Each cut operation requires selecting an eligible machine and its corresponding parameters, from a $k \times k$ grid. Between cuts, workers perform setup and transport operations on the machines. A correct schedule assigns the operations to the corresponding machines and workers while satisfying all constraints, such as preventing resource overlaps. Schedules are evaluated according to lexicographically prioritized objectives to be minimized: *(1)* job tardiness $Tard$, *(2)* total energy consumption TEC—including auxiliary, idle, and processing energy—, and *(3)* makespan C_{max}. Note that, as the machine parameters influence both processing time and energy consumption, optimizing such a problem is a highly complex task.

We map the objective values to the quality score z_j^q as follows. First, $Tard$, TEC, and C_{max} are normalized into $[0, 1]$ using the corresponding theoretical lower and upper bounds. Next, an exponential decay function is applied to compute the scores z_j^{Tard}, z_j^{TEC}, and $z_j^{C_{max}}$, which are subsequently combined into $z_j^q = POWA(z_j^{Tard}, z_j^{TEC}, z_j^{C_{max}})$.

5 Evaluation

In this section, we evaluate the proposed approach experimentally. [Semmelrock *et al.*, 2026] provides datasets, evolved programs, and results.

[3]https://www.voestalpine.com/

5.1 Datasets

For each case study, we use separate training and test sets, all comprising easy, moderate, and hard instances.

For the CCP, we reuse the dataset of Gebser *et al.* (2015), which comprises 99 instances: 30 easy, 33 moderate, and 36 hard, which, for brevity, we will denote by $30/33/36$. For the *training set*, we select 20 instances, split as $3/7/10$, as such a split was employed to define the CCP's competition instances for the 6th ASP Competition [Gebser *et al.*, 2017]. Moreover, we generate and include 5 more *very* easy instances (cf. Sec. 3.3) by reducing the number of components in the easy ones. Hence, the CCP's training set includes 25 instances $(8/7/10)$, while the *test set* comprises the 79 remaining ASP competition instances $(27/26/26)$.

For the HCP and the E-DFJSP, we generate both training and test sets, ensuring that all instances are satisfiable by design. The *training sets* are composed of 15 instances, split as $5/5/5$. For the HCP, the easy instances consist of up to 5 persons ($\sim$50 things), the moderate instances of up to 50 persons ($\sim$500 things), and the hard instances of up to 500 persons ($\sim$5 000 things). The hardness assessments reflect the results of [Semmelrock and Friedrich, 2025] on clingo's performance. Likewise, for the E-DFJSP, the easy instances involve up to 10 jobs ($\sim$60 operations), the moderate up to 50 jobs ($\sim$300 operations), and the hard up to 500 jobs ($\sim$3 000 operations). This is in accordance with [Schlenkrich and Parragh, 2022] where scheduling problems were classified as *large-scale* if they comprise at least 1 000 operations.

We designed the *test sets* to contain an overproportional number of hard instances ranging from large to *very* large size, in accordance with research question RQ3, i.e., investigating the scalability of the evolved code. Specifically, we used a split of $6/6/24$, where the 24 hard cases comprise up to 3 000 persons ($\sim$30 000 things) for the HCP and up to 1 000 jobs ($\sim$6 000 operations) for the E-DFJSP, thus going significantly beyond what is commonly already considered hard.

5.2 Experiments

Each experiment was conducted on a machine equipped with an AMD EPYC 7H12 64-core Processor, with RAM usage restricted to 32 GB. Timeouts for evolved programs and solvers to find a solution for any problem were set to 600 s.

Training. Since OpenEvolve+CHECKMATE is inherently non-deterministic, each evolutionary training run can produce a different best program p^*. To account for this variability, we performed four independent training runs for each case study (Sec. 4), yielding four corresponding best programs p^*.

Each problem has a specific prompt D and formal specification F. Exclusively for CCP, due to its hardness, *(i)* D includes the Failure Protocol (Sec. 3.2), and *(ii)* F is exploited to pinpoint violated constraints returned by the verifier whenever its verdict for a candidate solution equals *incorrect*.

While most parameters were kept at OpenEvolve's default values, the following problem-specific settings were used: *(i)* the number of evolutionary iterations (50 for HCP, 75 for CCP, 100 for E-DFJSP, or, for brevity $50/75/100$), *(ii)* the number of islands $(3/5/5)$, and *(iii)* the migration interval $(10/10/15)$ to promote greater diversity between

evolved programs. CHECKMATE's early stopping parameter k (Sec. 3.1) was set to 3. For all case studies, program evaluation criteria include built-in measures of code complexity and diversity, quantifying program length and (textual) differences. In addition, HCP's and CCP's criteria set contains the correctness, runtime, and memory usage scores. In contrast, E-DFJSP contains the quality and tardiness scores to guide the evolution more prominently towards programs that yield high-quality solutions. To balance performance, efficiency, and cost [Novikov *et al.*, 2025], we used as LLMs GPT-5 for 60% of queries and GPT-5-mini for the remaining 40%.

Testing. Each training run outputs a best program p^*, which was evaluated using CHECKMATE (standalone; without OpenEvolve) on the described test set (Sec. 5.1). To illustrate the performance variability across the four training runs, we focus on p^*_L and p^*_H. Here, p^*_L (p^*_H) denotes the evolved program that attains the lowest (highest) combined score on the test set, largely driven by the overall solving percentage.

Comparison with the baseline solvers and verifiers. To compare our evolved programs against state-of-the-art approaches, we executed top-performing solvers as standalone solving engines for each problem. For HCP and CCP, we employed clingo with the original formal specifications (HCP [Semmelrock and Friedrich, 2025]; CCP [Gebser *et al.*, 2017]), as it demonstrated strong solving performance in tests compared to a leading CP solver OR-Tools [Perron *et al.*, 2023]. For E-DFJSP, we used CP-Optimizer, following the results reported in [Da Col and Teppan, 2022], together with a formal specification extending that of Zuccato *et al.* (2025) in terms of a more general energy consumption formulation.

5.3 Synopsis of the Best Evolved Programs

The synopses of the best evolved programs were produced through a manual analysis conducted by the authors.

HCP. HCP's p^*_H begins by verifying available capacities by computing the minimum number of cabinets and rooms required under the ownership and capacity constraints and checks whether they exist. It proceeds greedily and deterministically, sorting things, cabinets and rooms and grouping them by owner, which enforces the ordering constraint. The program produces a correct configuration whenever sufficient resources exist, without exploring alternative assignments.

CCP. CCP's p^*_H combines heuristic-guided search with constraint-aware pruning and bounded backtracking to construct correct configurations over a state space of partial color, bin, and area assignments. Instead of exhaustively enumerating possibilities, the program incrementally builds a complete assignment while discarding infeasible branches early based on capacity, path, and area constraints, and applying deterministic ordering for repeatability. The procedure is not purely greedy—rejected assignments may be revisited through limited backtracking—yet it avoids full search by steering decisions via deterministic heuristics.

E-DFJSP. E-DFJSP's p^*_H consists of a greedy scheduler that first determines the set of eligible workers for each machine and, for every cut-machine pair, selects the best machine parameters by choosing the one with minimum processing time and, in case of ties, lower energy consumption. Jobs are ordered using the Earliest Due Date (EDD) heuristic to reduce total tardiness, and operations are scheduled respecting precedence constraints. Per task, all feasible machine-worker-mode combinations are evaluated to minimize total completion time, processing energy, and resulting machine makespan. The program maintains sorted calendars for machines, considering the interval from load start to unload completion, and for workers, who are required only during setup operations. Finally, it outputs a correct schedule together with the corresponding objective values.

Problem	HCP			CCP			E-DFJSP		
	p^*_L	p^*_H	SOL	p^*_L	p^*_H	SOL	p^*_L	p^*_H	SOL
Easy	100	100	100	63	93	100	100	100	100
Moderate	100	100	67	88	85	88	100	100	100
Hard	100	100	0	96	88	4	100	100	42
Overall	100	100	28	82	89	65	100	100	61

Table 1: Percentage of test instances solved by the evolved programs with the lowest and highest overall percentage (p^*_L and p^*_H) and by the state-of-the-art solver (SOL).

5.4 Results and Discussion

Tab. 1 summarizes the overall and difficulty-wise solving percentage for p^*_L, p^*_H, and the corresponding state-of-the-art solver (SOL) across the three selected case studies. For HCP and E-DFJSP, both p^*_L and p^*_H solve all instances, whereas for CCP they solve 82% and 89%, respectively. In contrast, SOLs perform well on easy instances but degrade substantially on the moderate and hard ones, achieving 28%, 65%, and 61% overall on HCP, CCP, and E-DFJSP, respectively. This indicates that both p^*_L and p^*_H are competitive and consistently outperform traditional solvers on harder instances, underscoring the scalability of our approach.

The cactus plots depicted in Figs. 2, 3, and 4 compare correctness, memory usage (top panel), and runtime (bottom panel) between p^*_H and SOL for each problem. Instances are ordered by increasing memory usage of SOL.

HCP. Fig. 2 shows a clear performance gap on HCP. While both approaches handle the easy instances, clingo's memory usage increases sharply at instance 11 and remains near saturation, leading to repeated timeouts (13 times), out-of-memory events (8 times), and internal errors (5 times) on all harder instances. In contrast, p^*_H maintains negligible memory (average: 281 MB) and runtime (average: 0.08 s) throughout and produces no incorrect solutions. Overall, p^*_H solves all instances with orders-of-magnitude lower resource usage, indicating superior scalability.

CCP. As can be seen from Fig. 3, p^*_H manifests consistently negligible runtimes while clingo exhibits three different runtime behaviors. More specifically, clingo solves 28 instances within 10 s and 23 instances between 10 s and the 10 min time limit, while 28 instances remain unsolved. In contrast, p^*_H solves 89% of the instances with consistently low memory and runtime, 546 MB and 0.12 s on average. Given this high performance, which applies to all four best programs, even a parallel portfolio solver combining all of them is practical

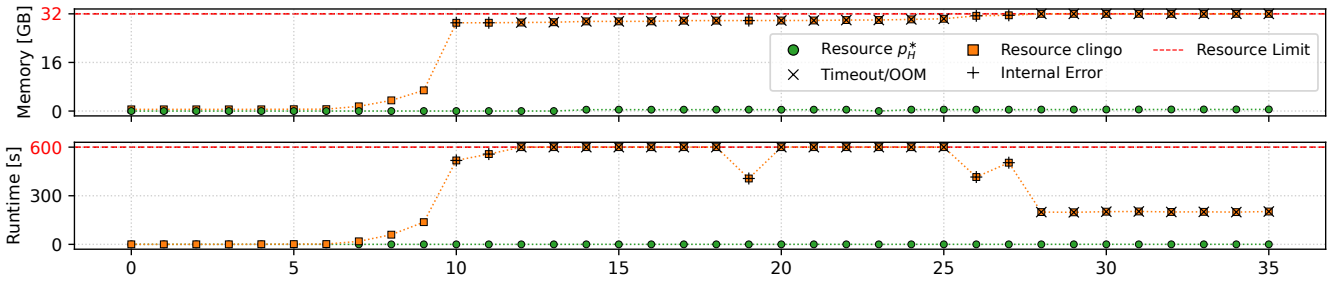

Figure 2: **HCP: Comparison between the evolved program (p_H^*) and clingo** wrt. memory $[GB]$ (above) and runtime $[s]$ (below). Symbols $\times, +$ depict the reason why clingo did not find a correct solution (OOM stands for out-of-memory). p_H^* is always correct.

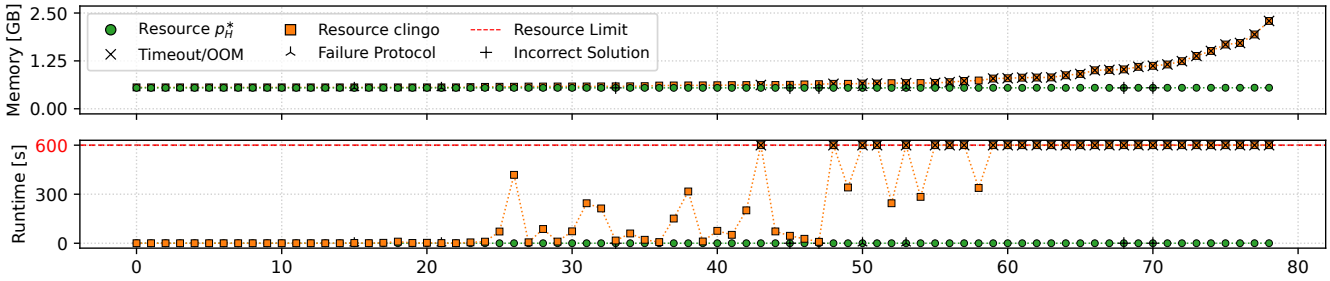

Figure 3: **CCP: Comparison between the evolved program (p_H^*) and clingo** wrt. memory $[GB]$ (above) and runtime $[s]$ (below). Symbol $\times$ depicts the reason why clingo did not find a correct solution (OOM stands for out-of-memory). The other symbols refer to p_H^*.

and achieves a 100% solving rate across all training and test instances. This yields two key findings: *(i)* the best program p_H^* solves more instances *and* exhibits a much lower resource usage than the state-of-the-art solver, and *(ii)* a portfolio of all four evolved programs can, to the best of our knowledge, *for the first time*, successfully solve all CCP instances from [Gebser *et al.*, 2015].

E-DFJSP. Fig. 4 compares p_H^* against CP Optimizer (CPO) for E-DFJSP. With regard to the solving rate, p_H^* is successful 100 % of the time on the test instances, whereas CPO fails to produce any solution on 14 of the 18 hardest ones due to 6 timeouts and 8 memory exhaustions. Regarding memory consumption, the two approaches exhibit comparable behavior on the *easy* and *moderate* instances; for the *hard* ones, however, CPO's space requirements escalate. In terms of runtime, p_H^* requires an average of 1.26 s across all instances to find a solution, whereas CPO either fails to produce any solution or achieves, if at all, only marginally better quality scores despite using the entire time budget. In fact, CPO finds the optimum only for the easiest instance (10 jobs, 1 machine). Notably, for all instances with 200 jobs or more, p_H^* outperforms CPO on tardiness—the objective of highest priority—by 49% on average. Furthermore, the state-of-the-art solver cannot solve most instances with at least 500 jobs within 10 min, while p_H^* always succeeds in less than 4 s.

Training and Checking Costs

The cost in API fees was at most € 20 per training run and below € 200 overall. Times per training run ranged from 1 to 16 hours, depending on the number of iterations and the efficiency of the generated programs. CHECKMATE's average and, respectively, worst-case costs for checking the correctness of outputs of best evolved programs p_H^* amounted to 3.01 s / 0.11 s / 1.93 s and 18.28 s / 0.32 s / 5.13 s per test instance for HCP / CCP / E-DFJSP.

6 Related Work

Combinatorial and optimization problems. Modern symbolic AI methods utilize declarative formalisms, such as Answer Set Programming [Gebser *et al.*, 2012], Constraint Programming [Rossi *et al.*, 2008], or Mixed Integer Programming [Wolsey, 2020], to address complex combinatorial and optimization problems. These approaches decouple problem specification from search algorithms, enabling domain-independent solvers to find optimal or near-optimal solutions using advanced techniques like conflict-driven clause learning and constraint propagation. However, large industrial problem instances significantly reduce solver performance in practice [Falkner *et al.*, 2018; Schlenkrich and Parragh, 2022]. To improve performance, research has focused on three directions [Kotary *et al.*, 2021]: *(i)* developing domain-specific heuristics, *(ii)* improving problem encodings, and *(iii)* configuring existing or learning new problem-specific algorithms. While designing any of these approaches requires significant domain expertise and manual effort, it has been observed that machine learning (ML) methods can simplify this challenge in many industrial cases where problem instances share similar patterns.

For the first direction, various ML techniques have been proposed to learn effective heuristics from, e.g., problem instances or solving traces, using supervised or reinforcement

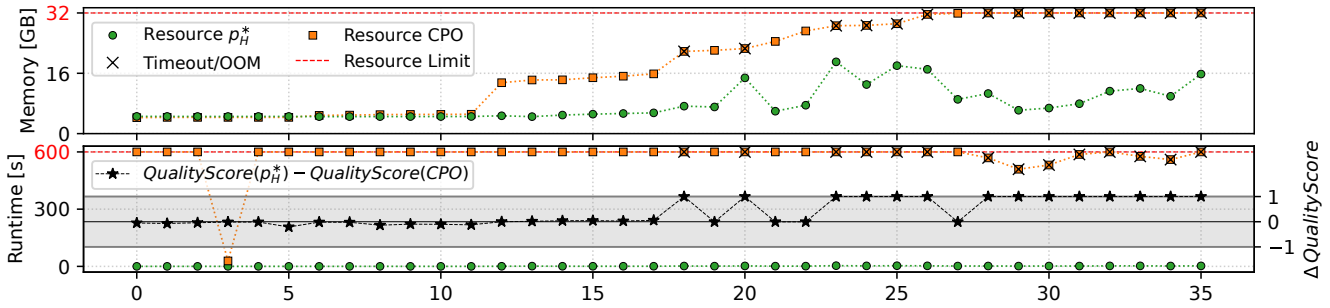

Figure 4: **E-DFJSP: Comparison between the evolved program (p_H^*) and CP Optimizer (CPO)** wrt. memory $[GB]$ (above), runtime $[s]$ (below, primary y-axis), and the difference on their quality score (below, secondary y-axis, gray background). If the difference is positive then p_H^* is better than CPO (when CPO does not find a solution, its quality score is 0). Symbol $\times$ depicts the instances where CPO did not find a correct solution (OOM stands for out-of-memory). p_H^* is always correct.

learning methods [Bengio *et al.*, 2021; Lodi and Zarpellon, 2017]. Injected into a solver, learned heuristics can significantly enhance its performance in the target domain. In the second direction, ML has been used to generate or optimize problem encodings, e.g., by adding symmetry-breaking or implied constraints [Tarzariol *et al.*, 2023; Taupe *et al.*, 2020], or finding problem decompositions [Cappart *et al.*, 2025]. In the third direction, ML approaches were first used to develop portfolio solvers capable of automatically configuring existing solvers for specific problem instances [Kotthoff, 2016]. More recently, deep learning methods have been proposed to solve combinatorial problems in an end-to-end fashion [Kotary *et al.*, 2021], learning to predict (approximate) solutions directly without invoking solvers at inference time.

Our approach can roughly be classified in the third direction, as we aim to automatically generate problem-specific solving algorithms. Unlike prior work that relies on deep learning models, we employ evolutionary computation combined with LLMs to synthesize code. This enables us to generate interpretable and verifiable algorithms, rendering our approach more flexible without compromising performance across different combinatorial and optimization problems.

Code generation for combinatorial optimization. Code evolution with LLMs has recently emerged as a particularly relevant research topic in combinatorial optimization. Early frameworks, such as FunSearch [Romera-Paredes *et al.*, 2024], focused on evolving heuristics for Cap Set and Bin Packing problems. Subsequent works, such as Evolution of Heuristics [Liu *et al.*, 2024], expanded the scope to a wider range of problems, such as online bin-packing, traveling salesman, and flow shop scheduling. Furthermore, AlphaEvolve [Novikov *et al.*, 2025] and its open-source implementations, such as OpenEvolve [Sharma, 2025], ShinkaEvolve [Lange *et al.*, 2025], DeepEvolve [Liu *et al.*, 2025], and CodeEvolve [Assumpção *et al.*, 2025], have applied code evolution to a variety of domains, focusing on the previously mentioned problems, as well as matrix multiplication, the minimum overlap problem, and kissing numbers in 11 dimensions. The aforementioned approaches neither explicitly mention formal verification nor utilize software testing, but rely on handcrafted solution checkers. More recently, the authors of [Georgiev *et al.*, 2025] applied AlphaEvolve to 67 mathematical problems, and, in a few cases, verified the program-generated solutions using AlphaProof [Hubert *et al.*, 2026] and Lean [de Moura *et al.*, 2015]. Notably, SATLUTION [Yu *et al.*, 2025] evolves entire code repositories to produce variants of SAT solvers, which operate at the propositional level. Differently, CHECKMATE employs first-order specifications to verify the correctness of candidate solutions for the given problem instances. Therefore, CHECKMATE addresses an open point in prior work by providing an automated declarative verification approach that guarantees the correctness of returned solutions.

7 Conclusions

The OpenEvolve+CHECKMATE framework showed the great potential to automatically synthesize problem-solving algorithms implemented in Python. The system needs no information on *how* to construct solutions. It relies only on *what* defines a correct (optimal) solution and a set of representative instances. The obtained programs can efficiently tackle large and challenging instances from the practical use cases of Siemens and voestalpine that are currently out of reach for state-of-the-art solvers. Our analysis demonstrates that the generated algorithms can successfully solve difficult real-world *(a)* combinatorial and *(b)* optimization problems such as configuration and scheduling, thereby addressing RQ1. The evolved programs significantly outperform state-of-the-art solvers on hard instances, addressing RQ2. Finally, the results highlight strong scalability: both in terms of increasing problem size, as tested on the HCP and E-DFJSP, and in terms of increasing problem hardness, as examined with regard to the CCP. This addresses RQ3 and confirms the practical viability of our approach.

Future research should explore several promising directions to further evaluate the approach and enhance its applicability and robustness. For example, by using CHECKMATE to verify the outputs of synthesized programs, we ensure solution correctness per instance, while establishing their overall correctness remains important future work. Moreover, we will conduct systematic ablation studies to quantify the contribution of every CHECKMATE component.

Acknowledgments

This research was funded in whole or in part by the Austrian Science Fund (FWF) 10.55776/COE12 and the Austrian Research Promotion Agency (FFG) FO999910235 (SAELING) and 930480ATRIA (ATRIA).

Contribution Statement

The first three authors contributed equally and are listed in alphabetical order. The same holds for the last three authors.

References

[Assumpção *et al.*, 2025] Henrique S. Assumpção, Diego Ferreira, Leandro Lacerda Campos, and Fabricio Murai. Codeevolve: An open source evolutionary coding agent for algorithm discovery and optimization. *CoRR*, abs/2510.14150, 2025.

[Bengio *et al.*, 2021] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *EJOR*, 290(2):405–421, 2021.

[Cappart *et al.*, 2025] Quentin Cappart, Tias Guns, Michele Lombardi, Gilles Pesant, and Dimos Tsouros. Combining constraint programming and machine learning: From current progress to future opportunities. *JAIR*, 84, 2025.

[Comploi-Taupe *et al.*, 2023] Richard Comploi-Taupe, Gerhard Friedrich, Konstantin Schekotihin, and Antonius Weinzierl. Domain-specific heuristics in answer set programming: A declarative non-monotonic approach. *JAIR*, 76:59–114, 2023.

[Da Col and Teppan, 2022] Giacomo Da Col and Erich C Teppan. Industrial-size job shop scheduling with constraint programming. *Oper. Res. Perspect.*, 9:100249, 2022.

[de Moura *et al.*, 2015] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Int'l Conf. on Automated Deduction (CADE)*, Lecture Notes in Computer Science, pages 378–388. Springer, 2015.

[Dodaro *et al.*, 2024] Carmine Dodaro, Giuseppe Galatà, Martin Gebser, Marco Maratea, Cinzia Marte, Marco Mochi, and Marco Scanu. Operating room scheduling via answer set programming: Improved encoding and test on real data. *JLC*, 34(8):1556–1579, 2024.

[El-Kholany *et al.*, 2025] Mohammed M. S. El-Kholany, Martin Gebser, and Konstantin Schekotihin. Decomposition strategies and multi-shot ASP solving for job-shop scheduling. *LMCS*, 21(3), 2025.

[Falkner *et al.*, 2016] Andreas A. Falkner, Gerhard Friedrich, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner. Twenty-five years of successful application of constraint technologies at Siemens. *AI Mag.*, 37(4), 2016.

[Falkner *et al.*, 2018] Andreas A. Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich Christian Teppan. Industrial applications of answer set programming. *KI*, 32(2-3):165–176, 2018.

[Fleischanderl *et al.*, 1998] Gerhard Fleischanderl, Gerhard Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intell. Syst.*, 13(4):59–68, 1998.

[Freuder, 2018] Eugene C. Freuder. Progress towards the holy grail. *Constraints*, 23(2):158–171, 2018.

[Friedrich *et al.*, 2011] Gerhard Friedrich, Anna Ryabokon, Andreas A. Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner. (re)configuration based on model generation. In *Second Workshop on Logics for Component Configuration*, volume 65 of *EPTCS*, pages 26–35, 2011.

[Gebser *et al.*, 2012] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool, 2012.

[Gebser *et al.*, 2015] Martin Gebser, Anna Ryabokon, and Gottfried Schenner. Solving combined configuration problems: a heuristic approach. In *Int'l Configuration Workshop*, CEUR Workshop Proceedings, pages 55–59. CEUR-WS.org, 2015.

[Gebser *et al.*, 2017] Martin Gebser, Marco Maratea, and Francesco Ricca. The sixth answer set programming competition. *JAIR*, 60:41–95, 2017.

[Gebser *et al.*, 2019] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele, and Philipp Wanko. *Potassco guide version 2.2.0*, 2019. Retrieved from https://github.com/potassco/guide/releases/tag/v2.2.0.

[Georgiev *et al.*, 2025] Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *CoRR*, abs/2511.02864, 2025.

[Gong *et al.*, 2018] Guiliang Gong, Qianwang Deng, Xuran Gong, Wei Liu, and Qinghua Ren. A new double flexible job-shop scheduling problem integrating processing time, green production, and human factor indicators. *J. Clean. Prod.*, 174:560–576, 2018.

[Hubert *et al.*, 2026] Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026.

[Kaufmann *et al.*, 2016] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and solving in answer set programming. *AI Mag.*, 37(3):25–32, 2016.

[Kotary *et al.*, 2021] James Kotary, Ferdinando Fioretto, Pascal Van Hentenryck, and Bryan Wilder. End-to-end constrained optimization learning: A survey. In *IJCAI*, pages 4475–4482. ijcai.org, 2021.

[Kotthoff, 2016] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data Mining and Constraint Programming*, volume 10101 of *LNCS*, pages 149–190. Springer, 2016.

[Laborie *et al.*, 2018] Philippe Laborie, Jerome Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP Optimizer for scheduling - 20+ years of scheduling with constraints at IBM/ILOG. *Constraints*, 23(2):210–250, 2018.

[Lange *et al.*, 2025] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *CoRR*, abs/2509.19349, 2025.

[Liu *et al.*, 2024] Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: towards efficient automatic algorithm design using large language model. ICML. JMLR.org, 2024.

[Liu *et al.*, 2025] Gang Liu, Yihan Zhu, Jie Chen, and Meng Jiang. Scientific algorithm discovery by augmenting alphaevolve with deep research. *CoRR*, abs/2510.06056, 2025.

[Lodi and Zarpellon, 2017] Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *TOP*, 25(2):207–236, 2017.

[Mouret and Clune, 2015] Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *CoRR*, abs/1504.04909, 2015.

[Novikov *et al.*, 2025] Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery. *CoRR*, abs/2506.13131, 2025.

[Perron *et al.*, 2023] Laurent Perron, Frédéric Didier, and Steven Gay. The CP-SAT-LP solver (invited talk). In *Int'l Conf. on Principles and Practice of CP, CP 2023*, LIPIcs, pages 3:1–3:2. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[Romera-Paredes *et al.*, 2024] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[Rossi *et al.*, 2008] Francesca Rossi, Peter van Beek, and Toby Walsh. Constraint programming. In *Handb. Knowl. Represent.*, volume 3 of *Foundations of Artificial Intelligence*, pages 181–211. Elsevier, 2008.

[Sanghikian *et al.*, 2026] Nathalie Sanghikian, Rafael Meirelles, Anand Subramanian, and Rafael Martinelli. A heuristic algorithm based on beam search and iterated local search for the maritime inventory routing problem. *Comput. Oper. Res.*, 188:107347, 2026.

[Schlenkrich and Parragh, 2022] Manuel Schlenkrich and Sophie N. Parragh. Solving large scale industrial production scheduling problems with complex constraints: an overview of the state-of-the-art. In *Int'l Conf. on Industry 4.0 and Smart Manufacturing*, Procedia Computer Science, pages 1028–1037. Elsevier, 2022.

[Semmelrock and Friedrich, 2025] Veronika Semmelrock and Gerhard Friedrich. Investigating the grounding bottleneck for a large-scale configuration problem: Existing tools and constraint-aware guessing. In *41st Int'l Conf. on Logic Programming*, EPTCS, pages 482–495, January 2025.

[Semmelrock *et al.*, 2026] Veronika Semmelrock, Benedetta Strizzolo, Francesco Zuccato, Gerhard Friedrich, Patrick Rodler, and Konstantin Schekotihin. Checkmate project. https://git-ainf.aau.at/checkmate/ijcai26, 2026.

[Sharma, 2025] Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent. https://github.com/algorithmicsuperintelligence/openevolve, 2025. Accessed: 23 September 2025.

[Tanese, 1989] Reiko Tanese. *Distributed genetic algorithms for function optimization*. PhD thesis, University of Michigan, USA, 1989.

[Tarzariol *et al.*, 2023] Alice Tarzariol, Martin Gebser, Konstantin Schekotihin, and Mark Law. Learning to break symmetries for efficient optimization in answer set programming. In *AAAI*, pages 6541–6549. AAAI Press, 2023.

[Taupe *et al.*, 2020] Richard Taupe, Antonius Weinzierl, and Gerhard Friedrich. Conflict generalisation in ASP: learning correct and effective non-ground constraints. *Theory Pract. Log. Program.*, 20(5):799–814, 2020.

[Wolsey, 2020] Laurence A Wolsey. *Integer programming*. John Wiley & Sons, 2020.

[Yager, 2009] Ronald R. Yager. Prioritized OWA aggregation. *Fuzzy Optim. Decis. Mak.*, 8(3):245–262, 2009.

[Yu *et al.*, 2025] Cunxi Yu, Rongjian Liang, Chia-Tung Ho, and Haoxing Ren. Autonomous code evolution meets np-completeness. *CoRR*, abs/2509.07367, 2025.

[Zuccato *et al.*, 2025] Francesco Zuccato, Patrick Rodler, Gerhard Friedrich, Konstantin Schekotihin, and Richard Comploi-Taupe. Energy-aware double-flexible job shop scheduling with machine modes and setup times: A real-world industrial case study using constraint programming. In *ECAI Workshop on AI-based Planning for Complex Real-World Applications (CAIPI 2025)*, CEUR Workshop Proceedings, pages 84–99. CEUR-WS.org, 2025.