On the use of ML and LLM for hard scheduling problems

Guillaume Povéda
Airbus SAS, Toulouse, France

Florent Teichteil-Koenigsbuch
Airbus SAS, Toulouse, France

Abstract

In an era where data-driven machine learning (ML), large language models (LLMs), and agentic AI are clearly improving efficiency in solving complex problems such as classifying images, answering questions, or playing complex games-solving NP-hard combinatorial problems remains largely dominated by long-developed methods like mathematical programming, constraint programming, or metaheuristics. In this preliminary work, we present two distinct approaches tested to tackle the RCPSP/Max problem, a classic scheduling benchmark that is notably challenging for mathematical programming to find even a feasible solution. One approach focuses on program synthesis using an evolutionary framework powered by an LLM, while the other is based on learning task ranking solutions from optimal solutions using graph neural networks (GNNs) coupled with a post-processing procedure to produce feasible schedules. We provide qualitative and empirical conclusions, discuss the true limits of LLM creativity and deep learning to solve challenging scheduling problems, and propose avenues for further work.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence

Keywords and phrases hybrid ML/OR, scheduling, program synthesis, generative AI

Digital Object Identifier 10.4230/LIPIcs...

Funding Our work has benefitted from the AI Interdisciplinary Institute ANITI. ANITI is funded by the French "Investing for the Future – PIA3" program under the Grant agreement n°ANR-23-IACL-0002.

1 Introduction

Hybridizing machine learning (ML) and classic operations research (OR) solvers has become a highly active area of research. While purely symbolic techniques like constraint programming (CP) have historically driven state-of-the-art results for constrained scheduling [8], recent breakthroughs in large language models (LLMs) suggest new avenues for discovering heuristics and programs automatically [7].

In this paper, we focus on the resource-constrained project scheduling problem with generalized precedence relations (RCPSP/Max) [4]. Finding even a feasible solution for highly constrained RCPSP/Max instances is notoriously difficult. We evaluate the potential of bridging ML and OR through two distinct experiments using benchmarks extracted from the well-known Kobe dataset:

1. **LLM program synthesis:** using the Shinka-Evolve framework to evolve algorithms to solve RCPSP/Max.
2. **Supervised GNN with post-processing:** learning from CP-SAT solutions and generalizing, followed by post-processing.

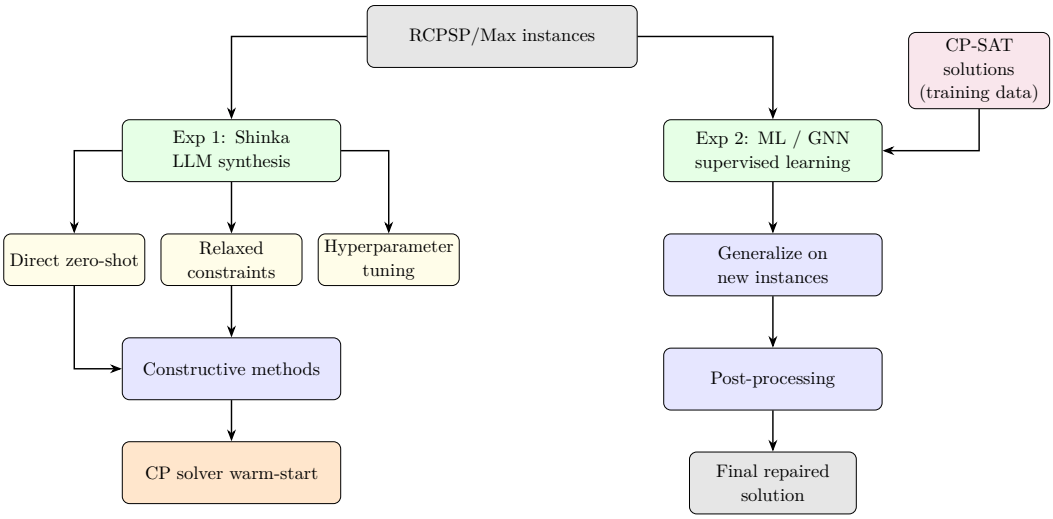

Figure 1 Overview of the experimental pipelines. Left: evolutionary program synthesis branching into constructive methods (used for CP warm-starting) and tuning. Right: supervised ML pipeline using CP-SAT data for training, followed by generalization and heuristic post-processing.

2 Experiment 1: program synthesis via LLMs

Our first experiment aims to test the capabilities of open-ended program evolution for combinatorial optimization. We utilized ShinkaEvolve [6], an LLM-driven evolutionary framework, to synthesize solver routines. The basic framework is similar to that of OpenEvolve [9] described in Figure 2. The framework operates as an asynchronous, closed-loop evolutionary system designed for maximum throughput. At its core, a central orchestrator continuously routes data between four distinct modules. First, a Prompt Sampler retrieves high-scoring historical programs from a Program Database to construct context-rich prompts. These prompts are dispatched to an LLM Ensemble, which acts as the evolutionary mutation operator to generate novel code modifications. The newly generated programs are then tested against the specific problem constraints by an Evaluator Pool. Finally, the valid programs and their fitness scores are archived back into the Program Database, closing the loop and providing a progressively improving context for subsequent generation cycles.

2.1 Methodology and results

We conducted several tests targeting the RCPSP/Max problem:

- **Direct zero-shot generalization:** We first attempted to generate solvers by evaluating LLM proposals on small-scale instances (without a CP solver in the loop) and testing their generalization to larger instances. This approach largely failed: While satisfiability of solutions rose above 50% on the training set (small instances), it was reduced to 0 for larger problem sizes.
- **Relaxed constraint solving:** We requested the LLM to solve a relaxed version of the problem where maximum time-lags were treated as soft penalties rather than hard constraints. The resulting scripts implemented a basic local search (hill climbing) with simple priority rules, and usually failed to reach a feasible solution.
- **Hyperparameter and technology tuning:** We tasked the LLM to act as a configuration agent, finding the best model mapping and hyper-parameters for any underlying

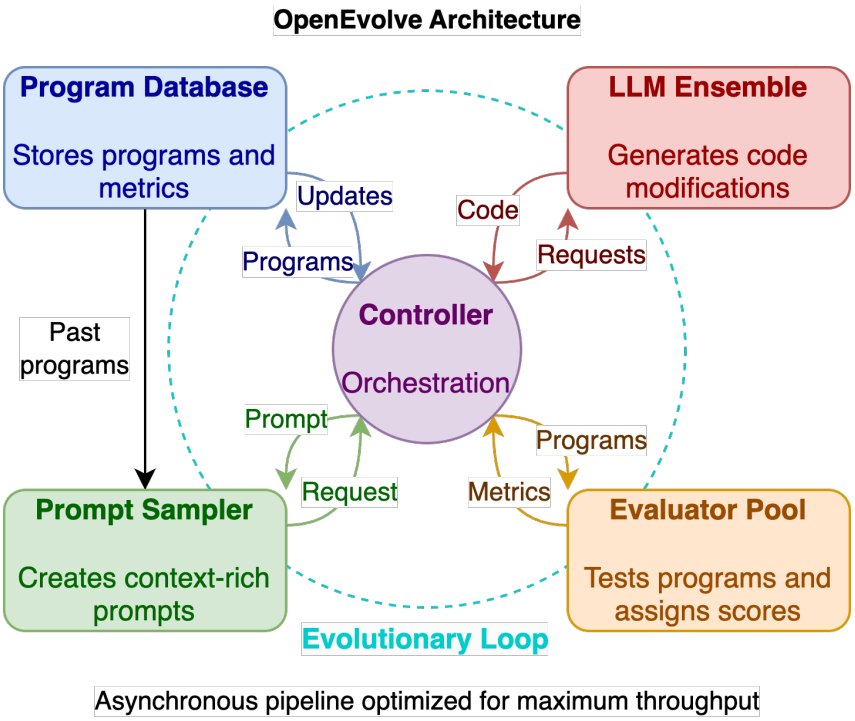

Figure 2 Open evolve framework (taken from 2)

technology solver. The final solver consisted of a CP solver formulating the problem with CP-SAT, leading to no advantage over our existing solver.

2.1.1 Convergence and introspection of the evolutionary framework

Shinka-Evolve comes with an advanced dashboard that allows us to carefully analyze how the code evolved. Here we analyze the results found from the *generalization* experiment. In particular, in Figures 3 and 4, we can see that the proportion of valid code increase through iterations, and observe that the cross operator has been the most efficient to improve the overall quality of the algorithm.

The evolutionary framework also calls the LLM to judge and describe the generated code, giving some insights in its reasoning. In these experiments, we obtained the following summary (LLM generated):

START-OF-SUMMARY

An analysis of the generation logs reveals that the LLM-driven evolutionary system incrementally reconstructed classical optimization methods.

- **Gen 0–23 (Naive to Greedy):** Transitioned from broken, hardcoded task placements to a standard, constraint-aware Serial Schedule Generation Scheme (SGS).
- **Gen 32–97 (Smart Heuristics):** Replaced simple ID-based sorting with intelligent, dynamic heuristics, prioritizing tasks by temporal slack and optimizing for Minimum Earliest Finish Time (MEFT).
- **Gen 144–202 (Exploration & GRASP):** Recognized stagnation in the greedy approach and introduced randomization via Restricted Candidate Lists (RCL) and multi-start mechanisms, effectively evolving an adaptive GRASP solver.

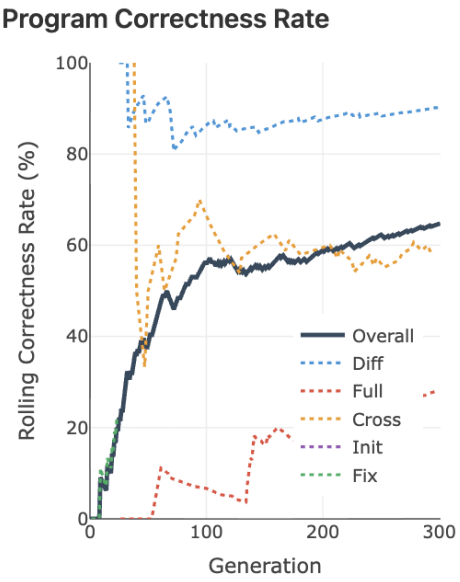

Figure 3 Evolution of correct programs rate (compiling, executing)

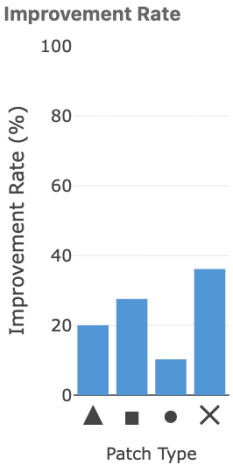

Figure 4 Improvement statistics per patch type (triangle: fix bugs, square: code diff, small changes, circle: full rewrite, cross: inspiration from different promising codes)

- **Gen 159 & 294 (Conflict Resolution):** Abandoned blind chronological backtracking in favor of dependency-directed "blame scoring" and lookahead penalties, learning to avoid moves that guarantee future deadlocks.
- **Gen 218–241 (Hybrid Intensification):** Integrated Large Neighborhood Search (LNS) with adaptive activation probabilities, allowing the algorithm to dynamically balance global exploration (GRASP) with targeted local intensification (LNS).

Core Insight: The evolutionary framework successfully learned to balance *exploration* and *exploitation*. It incrementally transformed a failing, naive script into a sophisticated hybrid metaheuristic by continuously identifying the root causes of infeasibility and stagnation

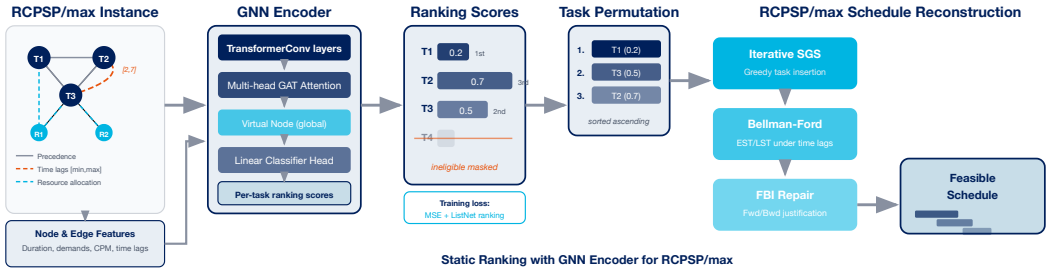

Figure 5 Full inference pipeline for extracting feasible schedules of RCPSP/max problems learned using Hetero Graph Neural Networks based on TransformerConv layers. This pipeline is also used during training to evaluate the currently learned model on a validation instance subset.

in previous generations.

END-OF-SUMMARY

2.1.2 Wrap-up

Results on this particular use case are quite negative at the moment, resulting in no generalization to larger instances for the first and second experiments, and no advantage over the existing CP model for the third. Some other experiments on other classes of problems where feasibility is easier to get, for example classical RCPSP or binpack could be interesting to investigate, with the disadvantage that existing constructive heuristics have been very well known for decades.

3 Experiment 2: GNN and learned heuristics

Inspired by recent successes in using GNNs for resource-constrained scheduling [10], our second experiment evaluates the potential of supervised learning by mapping the RCPSP/Max constraints into a graph structure. The full pipeline is depicted in Figure 5.

3.1 Supervised learning pipeline

Instead of relying on the LLM to write the core logic, we created a large training dataset consisting of optimal or high-quality solutions generated by the CP-SAT solver. We use the RCPSP/max instances from the Kobe dataset [1] which we split into 1169 instances for training and 1351 instances for testing, among which none of the 1000 task instances have been seen during training. A graph neural network was trained via supervised learning to predict job rankings extracted from CPSAT schedules, and then tasked with generalizing these predictions onto unseen instances.

3.2 Post-processing

Because the raw GNN predictions generated during the generalization phase often violate strict precedence or maximum time-lag constraints, we utilized LLM-generated code to build constructive post-processing routines designed to repair the GNN's output. The generated code begins with sorting learned normalized task ranking scores in ascending order to form a task permutation. This permutation is fed to an Iterative Serial SGS that greedily inserts tasks while respecting precedence and resource constraints, followed by a Bellman-Ford

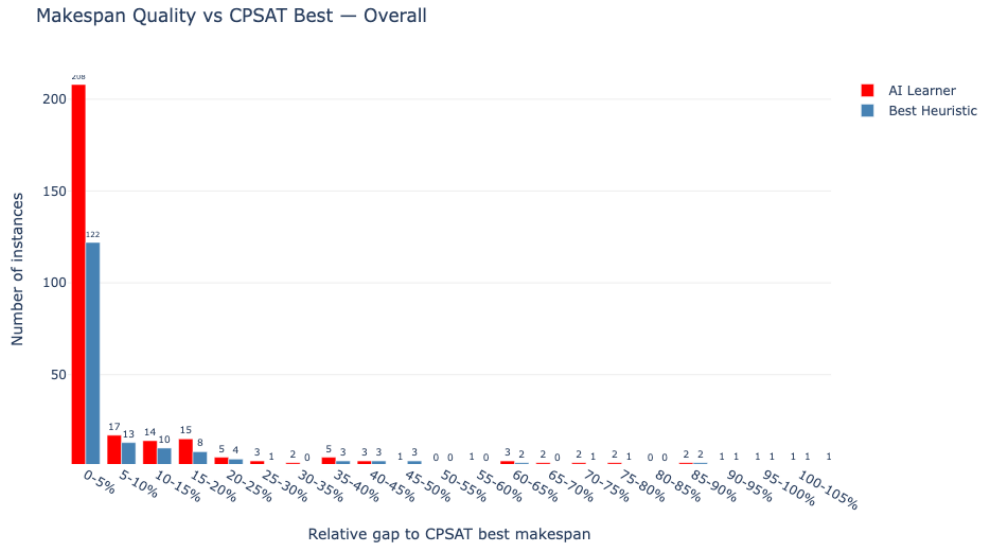

Figure 6 Number of RCPSP/max instances solved by the GNN Transformer and the Best Heuristic, sorted per makespan optimality gap 5% intervals relative to CPSAT's best solution (30 minutes timeout)

longest-path computation to enforce minimum and maximum time lag constraints on start times. Finally, a Forward-Backward Improvement (FBI) procedure alternates left- and right-justification passes to compress the schedule and repair any remaining constraint violations, yielding a feasible RCPSP/max schedule. The results were once again mixed. While the GNN could capture the rough topological ordering of the tasks, the "repaired" solutions frequently fell into local optima or failed validation checks, demonstrating that neural networks combined with basic post-processing still struggle with the strict logical feasibility demanded by RCPSP/Max.

3.3 Experiments

We compared our AI GNN learner to a set of well-known ranking heuristics to which we apply the same post-processing pipeline as for the AI learner in order to get feasible schedules from the rankings. Tested ranking heuristics are: minimum slack, Latest start time, Longest processing time, greatest resource demand, latest finish time, most total successors, greatest rank positional weight. Figure 6 shows the number of instances for which the AI learner and the best heuristic are within 5%-width makespan gap bins. We can see that the AI learner finds solutions of better quality than the best heuristic for each instance.

While the AI learner beats all the heuristics, it is not competitive against CPSAT. Indeed, in our experiments, CPSAT takes less computation time than the GNN pipeline to find a solution of a better quality on most instances. This result suggests that learning to replicate CPSAT solutions might be interesting for problems which are harder to solve for CPSAT. Following this observation, we applied the same learning framework and conducted experiments on multi-mode RCPSPs with non-renewable resources. Figure 7 shows a clear story: while CPSAT is mostly always faster than the GNN pipeline on RCPSP/max problems to get a solution of similar quality, it is the opposite on Multi-Mode RCPSPs for which CPSAT could not even find a solution as good as the AI learner within 30 minutes on 257

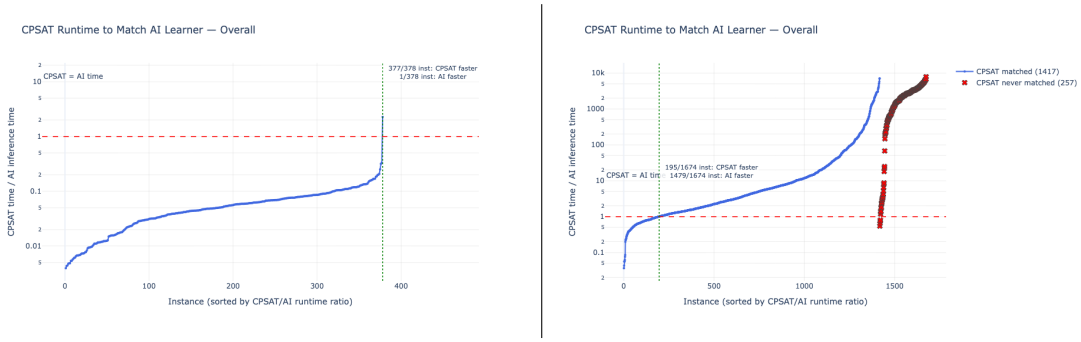

Figure 7 Relative runtime of CPSAT until it finds a solution of better quality than the GNN inference pipeline (the dashed horizontal line represents the GNN + Post-Processing runtime reference) for both RCPSP/max (left) and Multi-Mode RCPSPs (right).

instances.

4 Additional experiments

Following the acceptance of the paper in the workshop, we add more experiments in the appendix section, extending the program synthesis experiments to other class of problems.

4.1 Multimode RCPSP

We also tested both program synthesis 2 and the ML 3 methods to solve complex multi-mode RCPSP, from the MMLIB+ benchmark, introduced by [5]. We compare different solvers from the discrete-optimization library [2] on which we rely on for comparisons. The solvers are :

1. Constraint programming: 2 models based on 2 different backend solvers (Cpsat and Optal solvers)
2. Simulated annealing (using permutation based mutations)
3. A greedy solver (called Pile in the library), scheduling task with a priority list approach (2 methods)
4. The AI method from Section 3
5. A learned algorithm from Section 2

Again, we trained both AI and LLM method with a reduced set of MMLIB+ instances and then tested it on the 50 biggest instances. Results are detailed in Table 1. We let a time limit of 60 s per instance for each methods, but some methods would just provide one solution and won't improve (like the AI method and the Pile solver). We see that the AI method and the heuristic are not competitive with the final results of CP methods. However this conclusion is of course biased by the comparison with the final solution obtained by each method. Figure 8 details the solvers that dominate over each others for different time limits. The height of different stacked areas represents the proportion of instances where a given solver gets the best solution. We see that initially, the fastest solver gets best results (Pile and simulated annealing). Then around ∼1 second, the AI method becomes competitive, before CP gets advantage after a few seconds. Concerning the *Shinka*-heuristic, we should focus on the right figure which excludes the CP solver: we see that it starts to become competitive over the other methods. It slightly gains advantage over all the other heuristics approach after the 5-10 seconds zone. Actually the Shinka-heuristic is a local search algorithm, which

Table 1 Performance metrics and rank distributions across different configurations.

config	convergence time (s)	average relative overcost	average absolute overcost	Rank Distributions									
				#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
cpsat-auto2	28.25560229	0.007212757396	1.64	37	7	6	0	0	0	0	0	0	0
optal-baseline	38.72195275	0.03794555492	4.06	4	15	21	8	2	0	0	0	0	0
shinka-to-optal	59.39034708	0.04121621946	5.20	8	6	12	19	5	0	0	0	0	0
shinka-to-cpsat	58.75144765	0.0446031043	8.52	0	22	5	17	6	0	0	0	0	0
cpsat-baseline	42.24692066	0.1353752474	18.46	1	0	6	6	30	6	1	0	0	0
shinka-heuristic	45.90573013	0.2781173319	35.70	0	0	0	0	0	31	19	0	0	0
AI Method	0.876031965	0.3760439011	38.68	0	0	0	0	7	13	19	11	0	0
simulated-annealing	39.38971893	0.573023172	69.54	0	0	0	0	0	0	11	39	0	0
pile-most-successors	0.004437479239	1.882498973	189.60	0	0	0	0	0	0	0	0	44	6
pile-sample-successors	0.1157862833	1.980341536	203.54	0	0	0	0	0	0	0	0	6	44

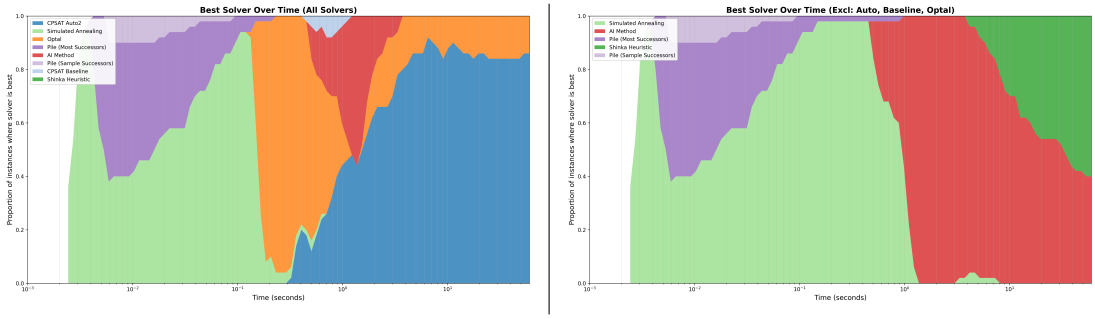

Figure 8 Best solver distribution, for given time limit

explains that it continuously improves but is not necessarily as fast as direct heuristics like AI and Pile.

From this experiment, we see some practical cases where the use of AI could be relevant : when the computation time is limited to <1s, for example in a scheduling under uncertainty setup where we have to massively generate schedules for many scenarios. For LLM, the exploration strategies seem relevant but in practice is slower than using CP, but the strategy could be adapted and directly used in the simulated annealing solver for example.

4.2 Bin packing with conflicts

Bin packing problems with conflicts is a variant of bin packing where each item i with weight w_i to be put in bins of capacity c can be constrained to not be in the same bin as some other items. We can see the bin packing problem as some scheduling problem (bin index representing time, and items being tasks to be accomplished). This is quite close to some other scheduling problem like simple assembly line balancing. Table 2 is detailing the results. This time, we can see that the generated heuristic is very competitive. Looking at the generated code, our conclusion is that it simply implemented a bit more advanced heuristic but close enough to state-of-the-art heuristics for this class of problem.

4.3 Sharing the code of the methods

When the source code will be in a ready state, it will be shared in the libraries we build upon: [2] and [3], where binding to program synthesis libraries have been implemented, and is also the base libraries used for the ML methods.

Table 2 Performance metrics and rank distributions across different configurations.

config	convergence time (s)	average relative overcost	average absolute overcost	Rank Distributions						
				#1	#2	#3	#4	#5	#6	# failed
shinka-to-cpsat	11.28553159	0	0	32	0	0	0	0	0	0
shinka-heuristic	0.007517152	0.003412162	0.303030303	23	9	0	1	0	0	0
cpsat-baseline-sched	13.38321029	0.008867833	0.428571429	4	3	0	0	0	0	26
cpsat-baseline-binary	13.01572614	0.011724976	0.571428571	3	3	0	1	0	0	26
optal-baseline	10.18577603	0.023712206	4.515151515	1	3	25	2	2	0	0
greedy-baseline	0.020535606	0.047849999	8.727272727	0	0	1	25	0	7	0

5 Lessons learnt and conclusion

These preliminary experiments provide several crucial insights into the current state of ML and LLMs for NP-hard scheduling. These points are certainly subject to discussion in the workshop, but here is what we found out so far:

- **The "creativity" of LLMs:** it remains difficult to make pure program synthesis work reliably for hard operations research problems. We must question and measure the real "creativity" of LLMs in this domain. Often, the models regurgitate well-known, classical constructive routines rather than inventing novel optimization algorithms.
- **The need for sound experiments:** robust evaluation is critical. While an LLM might successfully output Python code that runs, validating the logical soundness and optimality bounds on highly constrained benchmarks requires rigorous OR pipelines.
- **Problem class hardness:** the GNN experiments show that there is no point in learning to replicate a CP solver's solution if it takes milliseconds to the CP solver to find solutions of very good quality. Learning to generate full solutions to combinatorial optimization problems should be wisely investigated before going into it, especially for problem classes which are easy to solve for reasoning solvers like CP.
- **Future promise:** despite the mixed results on strict feasibility problems like RCPSP/Max, Program synthesis techniques using LLM appear highly promising for planning problems that possess a massive space of valid solutions (where optimization, rather than feasibility, is the bottleneck), as proved in the Beluga Challenge within the TUPLES project [11].

References

1. https://github.com/ptal/kobe-scheduling/tree/master/data/rcpsp-max.
2. Airbus. https://github.com/airbus/discrete-optimization, 2023.
3. Airbus. https://github.com/airbus/scikit-decide, 2022.
4. M. Bartusch, Rolf Möhring, and Franz Radermacher. Scheduling project networks with resource constraints and time windows. 16:199–240, 12 1988. doi:10.1007/BF02283745.
5. José Coelho and Mario Vanhoucke. Multi-mode resource-constrained project scheduling using rcpsp and sat solvers. *European Journal of Operational Research*, 213(1):73–82, 2011. URL: https://www.sciencedirect.com/science/article/pii/S037722171100230X, doi:10.1016/j.ejor.2011.03.019.
6. Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution, 2025. URL: https://arxiv.org/abs/2509.19349, arXiv:2509.19349.
7. Barekatain M. Novikov A. et al. Romera-Paredes, B. Mathematical discoveries from program search with large language models. *Nature*, 2024.

8 Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Solving the resource constrained project scheduling problem with generalized precedences by lazy clause generation, 2010. URL: https://arxiv.org/abs/1009.0347, arXiv:1009.0347.

9 Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL: https://github.com/algorithmicsuperintelligence/openevolve.

10 Florent Teichteil-Königsbuch, Guillaume Povéda, Guillermo González de Garibay Barba, Tim Luchterhand, and Sylvie Thiébaux. Fast and robust resource-constrained scheduling with graph neural networks. *Proceedings of the International Conference on Automated Planning and Scheduling*, 33(1):623–633, Jul. 2023. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/27244, doi:10.1609/icaps.v33i1.27244.

11 TUPLES. https://tuples.ai/2025/10/04/beluga-award-ceremony/.