

Can LLMs Build a MaxSAT Solver from Papers? The CoreForge Experience

Ruben Martins   

Carnegie Mellon University

Abstract

We report on CoreForge, an experience in using large language models (LLMs) to build an unweighted MaxSAT solver from research papers rather than from an existing solver codebase. The project focuses on unsatisfiability-based MaxSAT algorithms and follows an iterative workflow that combines paper discussions with ChatGPT, implementation through Codex prompts, and repeated LLM-assisted code audits and revisions. Although the codebase implements several algorithms and solver components, our evaluation focuses on configurations that combine core-guided optimization, lightweight preprocessing, core minimization, integration with integer linear optimization backends, and a new core-sequence lookahead approach.

Our experience suggests that LLMs can support solver implementation from papers, while requiring external validation, benchmarking, and human guidance. In our experiments, fuzzing and MaxSAT Evaluation instances did not reveal wrong answers in the tested configurations, although performance remains below the best hand-engineered MaxSAT solvers. We summarize what worked, what remained difficult, and the lessons for future LLM-assisted solver development.

2012 ACM Subject Classification Software and its engineering → Automatic programming; Theory of computation → Automated reasoning; Theory of computation → Constraint and logic programming

Keywords and phrases MaxSAT, LLMs, Code Generation

Funding This work was partially supported by the National Science Foundation (NSF) under Award CCF2427581 and DARPA under Agreement FA8750-24-9-1000.

1 Introduction

Large language models (LLMs) can generate substantial software artifacts, but their ability to support the development of constraint solvers remains less clear. Solver development is a demanding test case because competitive solvers combine algorithmic ideas with careful implementation, preprocessing, heuristics, data structures, and extensive performance tuning.

Recent work has explored LLMs for solver engineering, including heuristic optimization for existing SAT solvers, repository-scale SAT solver evolution, SAT-solving code generation, and SMT solver construction [17, 18, 14, 16, 6]. In parallel, paper-to-code benchmarks and systems study whether AI agents can implement research ideas from papers [15, 13, 4]. CoreForge connects these directions by studying whether LLMs can build a MaxSAT solver from research papers rather than from an existing solver codebase.

This distinction is important because most solver-focused systems either start from existing SAT solver implementations, optimize selected components, or target restricted SMT fragments. In contrast, CoreForge studies a longer-horizon paper-to-code task in which ideas from multiple MaxSAT papers are turned into a growing solver codebase, revised over many iterations, and evaluated on standard MaxSAT benchmarks. The result is not a solver-evolution study over an existing implementation, but a case study in iterative LLM-assisted solver construction from papers.

This experience addresses three questions: (1) can LLMs build a nontrivial MaxSAT solver from papers, (2) were wrong answers observed, and (3) how far is the resulting performance from hand-engineered MaxSAT solvers?

2 The CoreForge Experience

CoreForge was developed as an iterative paper-to-code experiment, not a one-shot generation task. Each iteration translated MaxSAT ideas into prompts, integrated generated changes, and used audits, fuzzing, and benchmarks to guide revisions.

Can LLMs build a MaxSAT solver from papers?

In our experience, yes, but not in a one-shot or fully autonomous way. CoreForge was built from scratch rather than from an existing MaxSAT solver codebase. The human role was to select papers, define implementation goals, run experiments, and decide which generated changes to keep. The LLM produced the main solver architecture and implemented several unsatisfiability-based MaxSAT algorithms, including PM2 [3], MSU3 [7], and OLL [8], together with lightweight preprocessing [5], core minimization [5], integration with optimization backends such as SCIP [1] and CP-SAT [11], and a new core-sequence lookahead approach inspired by recent work on OLL reformulation structure and core sequences [12, 2]. The solver runs bounded probes with different core-extraction strategies, scores the resulting core-sequence prefixes, and then commits to the most promising strategy for the main search. The final codebase contains more than 25K lines of C++ code and includes algorithms and components beyond those enabled in our evaluation. However, the process required repeated prompting, benchmarking, and LLM-assisted auditing. The LLM was effective at translating high-level algorithmic ideas into code, but less effective at aggressive low-level solver engineering. Appendix A gives additional details on the development workflow.

Were wrong answers observed?

We evaluated the tested configurations using fuzzing [10] and MaxSAT Evaluation instances, and did not observe wrong answers. This was surprising given the size of the generated codebase and the fact that we intentionally treated the implementation largely as a black box, relying on testing, benchmarking, and LLM-assisted audits rather than detailed manual code inspection. This evidence should be interpreted cautiously. During development, fuzzing and LLM-assisted audits uncovered issues that were later fixed by the coding agent, underscoring the need for external validation.

How competitive are the solvers?

CoreForge solves a substantial number of nontrivial MaxSAT Evaluation instances, although it is not yet competitive with the best hand-engineered MaxSAT solvers. We evaluated three configurations: a core-guided OLL baseline, a variant with integer-linear-optimization backend integration, and a variant with core-sequence lookahead. While the individual configurations remain below the strongest solvers, they outperform several established solvers on our benchmark subset. Their virtual best solver suggests that better instance-level selection could further improve performance. Appendices C and D give more details.

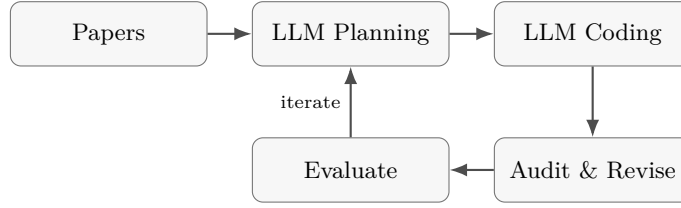
3 Conclusion and Future Work

CoreForge suggests that LLMs can support substantial solver construction from research papers when paired with iterative guidance, benchmarking, and external validation. The main lesson is that LLM-assisted paper-to-code workflows can already support solver development, but trust and autonomy remain central challenges.

Future work should study more autonomous, self-improving solver agents that can read papers, propose implementations, run benchmarks, audit failures, and revise their own code. A central challenge is ensuring that they implement the intended algorithms, rather than producing code that merely works on the available benchmarks.

References

- 1 Tobias Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009. doi:10.1007/s12532-008-0001-1.
- 2 Josep Alòs, Carlos Ansótegui, and Eduard Torres. Revisiting sat-based solvers: Maxsat rules and core sequences. *Journal of Artificial Intelligence Research*, 85:Article 16, 2026. doi:10.1613/jair.1.19525.
- 3 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Solving weighted partial maxsat through satisfiability testing. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 5584 of *Lecture Notes in Computer Science*, pages 427–440. Springer, 2009.
- 4 Tianyu Hua, Harper Hua, Violet Xiang, Benjamin Klieger, Sang T. Truong, Weixin Liang, Fan-Yun Sun, and Nick Haber. ResearchCodeBench: Benchmarking LLMs on Implementing Novel Machine Learning Research Code. *CoRR*, abs/2506.02314, 2025. arXiv:2506.02314.
- 5 Alexey Ignatiev, António Morgado, and Joao Marques-Silva. RC2: An efficient maxsat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, 2019. doi:10.3233/SAT190116.
- 6 Mikoláš Janota and Mirek Olsák. LLM2SMT: Building an SMT Solver with Zero Human-Written Code. *CoRR*, abs/2603.06931, 2026. arXiv:2603.06931.
- 7 Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for maxsat. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2014. doi:10.1007/978-3-319-10428-7_39.
- 8 António Morgado, Carmine Dodaro, and Joao Marques-Silva. Core-guided maxsat with soft cardinality constraints. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, 2014. doi:10.1007/978-3-319-10428-7_42.
- 9 Miguel Neves, Ruben Martins, Mikoláš Janota, Inês Lynce, and Vasco Manquinho. Exploiting resolution-based representations for maxsat solving. In *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2015. doi:10.1007/978-3-319-24318-4_20.
- 10 Tobias Paxian and Armin Biere. Uncovering and classifying bugs in maxsat solvers through fuzzing and delta debugging. In *Proceedings of the 14th International Workshop on Pragmatics of SAT (POS 2023)*, volume 3545 of *CEUR Workshop Proceedings*, pages 59–71. CEUR-WS.org, 2023.
- 11 Laurent Perron and Vincent Furnon. The CP-SAT-LP solver. In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:2. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CP.2023.3.
- 12 André Schidler and Stefan Szeider. Analyzing reformulation performance in core-guided maxsat solving. In *28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025)*, volume 341 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.SAT.2025.26.
- 13 Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2Code: Automating Code Generation from Scientific Papers in Machine Learning. *CoRR*, abs/2504.17192, 2025. arXiv:2504.17192.
- 14 Junjie Sheng, Yanqiu Lin, Jiehao Wu, Yanhong Huang, Jianqi Shi, Min Zhang, and Xiangfeng Wang. SolSearch: An LLM-Driven Framework for Efficient SAT-Solving Code Generation. In *International Conference on Software Engineering: New Ideas and Emerging Results, ICSE 2025 - NIER*, pages 6–10. IEEE, 2025.



■ **Figure 1** High-level iterative workflow used to develop CoreForge. Papers were used for planning, planning was converted into implementation prompts, and the resulting code was evaluated, audited, and revised across iterations.

- 15 Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. PaperBench: Evaluating AI’s Ability to Replicate AI Research. *CoRR*, abs/2504.01848, 2025. [arXiv:2504.01848](#).
- 16 Yiwen Sun, Furong Ye, Zhihan Chen, Ke Wei, and Shaowei Cai. Automatically discovering heuristics in a complex SAT solver with large language models. *CoRR*, abs/2507.22876, 2025. [arXiv:2507.22876](#).
- 17 Yiwen Sun, Furong Ye, Xianyin Zhang, Shiyu Huang, Bingzhen Zhang, Ke Wei, and Shaowei Cai. AutoSAT: Automatically Optimize SAT Solvers via Large Language Models. *CoRR*, abs/2402.10705, 2024. [arXiv:2402.10705](#).
- 18 Cunxi Yu, Rongjian Liang, Chia-Tung Ho, and Haoxing Ren. Autonomous Code Evolution Meets NP-Completeness. *CoRR*, abs/2509.07367, 2025. [arXiv:2509.07367](#).

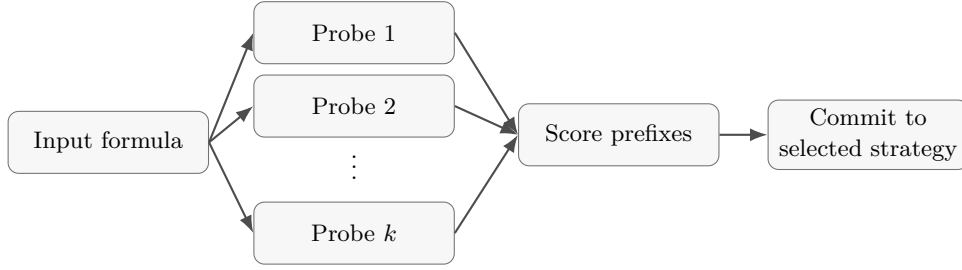
A Development Workflow

This appendix summarizes the workflow used to develop CoreForge. The goal was to study whether a MaxSAT solver could be built from research papers, rather than from an existing solver codebase. Accordingly, research papers were the primary technical input, and source code from other MaxSAT solvers was not analyzed.

Figure 1 shows the overall workflow. The process began by providing papers to ChatGPT one by one, using the discussions to extract the main algorithmic ideas, identify promising features, and plan implementation steps. These planning discussions were then converted into concrete prompts for Codex, which generated or revised solver code. The resulting code was audited with ChatGPT and Codex to identify mismatches between the intended algorithm and the implementation, missing corner cases, and opportunities for revision. This process was repeated throughout development.

A practical motivation for separating planning from coding was cost control. ChatGPT was used primarily for higher-level planning and design discussions, while Codex was reserved for implementation tasks. Fuzzing and benchmark evaluation were run manually, rather than delegated to the agent, to keep control over the validation process and avoid spending Codex time on long experimental runs.

The development spanned approximately four weeks and largely exhausted the weekly Codex usage budget in the \$20 plan. Most of the work used GPT-5.4, while GPT-5.5 became available near the end of the project. Overall, the process involved more than 70 large prompts, around 120 Git commits, and analysis of roughly 30 MaxSAT papers together with MaxSAT Evaluation proceedings. The commits were created manually after incorporating LLM-generated changes. No external solver code was analyzed, since the purpose of the experiment was to study a paper-to-code workflow.



■ **Figure 2** High-level structure of core-sequence lookahead. Several bounded probes explore alternative early core sequences; the solver scores the observed prefixes and commits to the most promising strategy.

The difficulty of implementation varied substantially across features. Some components, such as incremental totalizer encodings [7], required many iterations before they were integrated correctly. Other ideas were implemented only partially or differently from the original papers, as happened with some partitioning-based techniques [9]. As the codebase grew, integration also became harder. Adding new features, refactoring existing components, and removing unused code increasingly consumed a significant fraction of the agent’s effort.

In addition to code generation, the LLMs were used for implementation audits. This was important because we did not manually inspect the generated code. Instead, we treated the solver largely as a black box and relied on LLM-assisted audits, manually run fuzzing, and manually run benchmark evaluation to identify correctness and performance issues.

B New Feature Development: Core-Sequence Lookahead

One part of the CoreForge development process went beyond reimplementing existing algorithms and explored whether LLMs could help implement new features. The resulting approach, called *core-sequence lookahead*, was motivated by recent work showing that the trajectory of a core-guided MaxSAT solver matters. Schidler and Szeider analyze OLL reformulations through the structure of the generated cores [12], while Alòs, Ansótegui, and Torres show that different core sequences can lead to substantially different solving behavior [2]. Inspired by these observations, core-sequence lookahead explores several early core sequences before committing to the main search.

The goal of core-sequence lookahead is to use a small, bounded probing phase to compare alternative early search trajectories before committing to the main run. Each probe follows a different strategy for ordering assumptions and extracting or processing cores. The solver records the resulting core-sequence prefix, scores it using inexpensive statistics, and selects the most promising strategy for the subsequent search. Thus, this approach turns variation in early core discovery into an explicit, bounded selection step.

Figure 2 shows the high-level structure. The solver first runs several bounded probes. Each probe attempts to extract a short sequence of cores under a SAT-call and conflict budget. The resulting sequences are scored using inexpensive statistics, such as lower-bound gain, core sizes, and the number of distinct soft clauses involved. The selected strategy is then used for the subsequent MaxSAT search.

The feature was implemented through the same LLM-assisted workflow used for the rest of the solver. The high-level idea was discussed with ChatGPT and then converted into a Codex implementation prompt. Codex added the lookahead infrastructure, including strategy definitions, probe budgets, prefix statistics, scoring, and integration with the main solver

■ **Table 1** CoreForge components in the three evaluation configurations.

Component	Baseline	ILP	Lookahead
OLL Core-guided Algorithm [8]	✓	✓	✓
Intrinsic AtMost1 preprocessing [5]	✓	✓	✓
Disjoint-core preprocessing [3]	✓	✓	✓
Core minimization [5]	✓	✓	✓
Initial upper bound		✓	✓
SCIP presolve/search [1]		✓	✓
CP-SAT backend [11]		✓	✓
Core-sequence lookahead			✓

configuration. Subsequent LLM-assisted audits focused on implementation risks specific to lookahead, including whether probe state was kept separate from the committed run, whether budgets were enforced consistently, and whether the selected strategy was actually used after probing.

In the evaluated lookahead configuration, CoreForge runs eight bounded probes before the main MaxSAT search. Each probe has limits on the number of SAT calls, conflicts per call, and cores collected. After this bounded exploration, the solver selects the most promising strategy and restarts the main search using that strategy.

This appendix does not present core-sequence lookahead as a finished MaxSAT technique. As discussed in the evaluation (Appendix D), the current implementation still has limitations. Instead, the feature serves as an example of a different use of LLMs in solver development. The LLM was not only asked to implement known algorithms from papers, but also to help turn a research intuition into a concrete solver feature, integrate it into a growing codebase, and revise it through audit and benchmarking.

C Evaluation Configurations

This appendix summarizes the three CoreForge configurations used in the evaluation. The configurations are summarized in Table 1. All configurations target unweighted MaxSAT and share the same core-guided MaxSAT infrastructure. These configurations are being prepared for submission to the MaxSAT Evaluation 2026.

- **Baseline.** The baseline configuration measures the performance of the LLM-generated MaxSAT solver without external optimization backends. It uses an OLL-style unsatisfiability-based algorithm [8], with incremental construction of cardinality constraints [7], as the main optimization procedure. The configuration also enables lightweight preprocessing, including intrinsic AtMost1 reasoning over soft clauses [5], disjoint-core preprocessing [3], and bounded core minimization [5].
- **ILP.** The ILP configuration extends the baseline with optimization-backend integration. It first computes an initial upper bound by finding a feasible model and then applying a bounded local-improvement phase that perturbs the model and keeps assignments that reduce the number of unsatisfied soft clauses. The resulting upper bound is passed to the subsequent optimization stages. The configuration also runs SCIP [1] for presolving and solving, and CP-SAT [11] as an additional optimization backend, each with a short timeout. When these backend stages do not solve the instance, the remaining search is handled by the core-guided MaxSAT engine.

- **Lookahead.** The lookahead configuration extends the ILP configuration with core-sequence lookahead, described in Appendix B. Before the main MaxSAT search, the solver runs a bounded probing phase with several strategies for ordering assumptions and processing early cores. The observed prefixes are scored using lightweight statistics, and the selected strategy is used for the subsequent search.

D Performance Results

This appendix reports the performance of the three CoreForge configurations used in our evaluation. The goal of these experiments is not to claim that CoreForge improves the state of the art, but to assess the current implementation, identify its main bottlenecks, and quantify the potential impact of the proposed extensions.

Experimental setup

We evaluated the baseline, ILP, and lookahead configurations on a 417-instance subset of the MaxSAT Evaluation 2024 benchmarks. Since the goal is a diagnostic comparison rather than a full competition-scale evaluation, we focus on instances that were solved by at least one solver in the MaxSAT Evaluation 2024 and whose optimum value is larger than zero. This selection gives a nontrivial benchmark subset with known reference optima, while avoiding instances that are either unsolved by all evaluated solvers or have a zero-cost optimum.

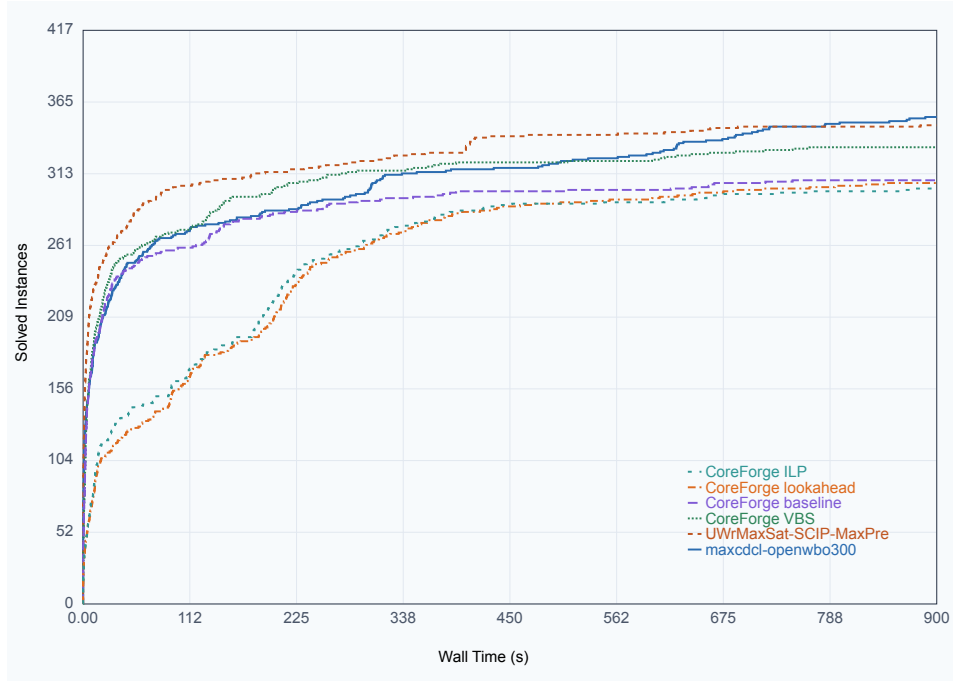
Overall performance

Figure 3 shows a cactus plot comparing the three CoreForge configurations and their VBS against `maxcdcl-openwbo300` and `UWrMaxSat-SCIP-MaxPre`. The plot reports, for each solver, the number of instances solved within a given runtime budget.

The cactus plot shows that the current CoreForge configurations still lag behind the strongest solvers, both in the number of solved instances and in the rate at which instances are solved. This is expected, since CoreForge is still an early implementation and several of its components are not yet guarded by robust instance-level resource policies. In particular, the ILP and lookahead configurations can introduce overhead on instances where their additional reasoning is not beneficial. This explains why these configurations do not strictly dominate the baseline, even though they solve additional instances on some benchmark families.

Beyond the aggregate runtime curves, the per-instance comparison shows that CoreForge is not merely solving a subset of the instances solved by existing solvers. The CoreForge VBS solves instances missed by strong baselines such as `UWrMaxSat-SCIP-MaxPre` and `maxcdcl`. In particular, CoreForge solves 9 instances that are not solved by `UWrMaxSat-SCIP-MaxPre` and 19 instances that are not solved by `maxcdcl`.

Conversely, the instances solved by these solvers but missed by CoreForge highlight the main remaining weaknesses of the current implementation. `UWrMaxSat-SCIP-MaxPre` solves 25 instances that are not solved by CoreForge, 13 of which belong to the `frb` family. Similarly, `maxcdcl` solves 41 instances that are not solved by CoreForge, with 35 of them concentrated in two families, namely 13 in `frb` and 22 in `minimize-5gons`. This concentration suggests that the gap to the strongest solvers is not uniformly spread across the benchmark set. Instead, a small number of benchmark families account for a large fraction of the remaining gap. Improving CoreForge on these families could therefore bring its performance substantially closer to the state-of-the-art solvers.



■ **Figure 3** Cactus plot comparing the baseline, ILP, and lookahead configurations, and their VBS against two strong solvers from the unweighted track of the MaxSAT Evaluation 2024.

Comparison against other MaxSAT Evaluation 2024 solvers

Table 2 compares CoreForge against the strongest solvers on the selected 417-instance subset. The results should be interpreted with care because they combine runs obtained in two environments. Some solvers were rerun on our local machines, while the remaining results are taken from the MaxSAT Evaluation 2024. The locally rerun baselines provide a useful calibration point. In particular, `maxcdcl-openwbo300` solves the same number of instances locally as in the evaluation data, even though our runs used a 900 s timeout while the MaxSAT Evaluation 2024 used a 1200 s timeout. Moreover, `UWrMaxSat-SCIP-MaxPre` solves slightly more instances in our local setting. These results suggest that the local runs are useful for identifying broad performance trends, although the comparison is not a fully controlled head-to-head evaluation.

The current individual CoreForge configurations are not yet competitive with the best solvers in total number of solved instances. The baseline solves 308 instances, while the lookahead and ILP configurations solve 306 and 302 instances, respectively. However, the CoreForge virtual best solver (VBS) solves 332 instances, substantially more than any individual CoreForge configuration. This gap shows that the three configurations are complementary, since each configuration solves instances that the others miss. It also suggests that much of the gap to the VBS is recoverable.

The main limitation is overhead control. The ILP and lookahead configurations extend the baseline, but in the current implementation they can lose instances when their additional reasoning is not beneficial. The ILP configuration solves 14 instances that the baseline misses, but loses 20 instances, mostly because SCIP or CP-SAT can become too expensive on large formulas. Similarly, the lookahead configuration solves 11 instances that the ILP configuration misses, but loses 7 instances because the lookahead phase can take too long.

■ **Table 2** Number of instances solved within the timeout on the selected 417-instance subset.

Solver	Source	Solved
maxcdcl-openwbo300	CF	354
maxcdcl-openwbo300	MSE'24	354
maxcdcl	CF	348
UWrMaxSat-SCIP-MaxPre	CF	348
UWrMaxSat-SCIP-MaxPre	MSE'24	341
EvalMaxSAT-SBVA	MSE'24	340
CoreForge VBS	CF	332
EvalMaxSAT-SBVA-saveCores	MSE'24	329
EvalMaxSAT-SBVA_1	MSE'24	328
wmaxcdcl	MSE'24	317
wmaxcdcl-openwbo1200	MSE'24	316
CoreForge baseline	CF	308
CoreForge lookahead	CF	306
CoreForge ILP	CF	302
Pacose	MSE'24	292
PacoseMP2	MSE'24	288
Exact	MSE'24	274
CASHWMaxSAT-DisjCad-S15	MSE'24	248
CASHWMaxSAT-DisjCad-S18	MSE'24	248
CASHWMaxSAT-DisjCom-S15	MSE'24	240
CASHWMaxSAT-DisjCom-S18	MSE'24	240

CF results were obtained on our local machines, each with 32 cores, 64 GB RAM, and an Intel Xeon Silver 4110 CPU at 2.10 GHz. Local runs used a 16 GB memory limit and a 900 s timeout. MSE'24 results are taken from the MaxSAT Evaluation 2024, which used a 1200 s timeout. Since the results were obtained on different machines and with different time limits, the comparison should be interpreted as approximate rather than as a fully controlled head-to-head evaluation. The CoreForge VBS row is the virtual best over the baseline, ILP, and lookahead configurations.

In the current implementation, lookahead is controlled by SAT-conflict limits, but these limits do not always bound wall-clock overhead. These losses are therefore not fundamental limitations of the techniques, but missing guardrails in the current implementation.

Overall, the VBS result gives a concrete target for the next implementation iteration. Since ILP and lookahead build on top of the baseline, conservative instance-level policies should allow CoreForge to keep most of their gains while avoiding cases where their overhead dominates. For ILP, the solver should skip SCIP and CP-SAT, or give them a much smaller budget, on formulas that are too large or when early presolve does not produce enough simplification. For lookahead, a direct wall-clock budget should be added in addition to the existing per-call conflict limits. These changes should move the best individual configuration substantially closer to the VBS. Moreover, even with these current limitations, CoreForge already solves more instances than several established MaxSAT solvers in the comparison.