# KissatEvolve: Controllable and Scalable Synthesis of SAT Heuristics with Densely Annotated Memory Bank

**Meru Gopalan**[1]
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

**Jialin Lu**[2]
School of Computing Science, Simon Fraser University, Burnaby, BC, Canada

**Jialin Song**[2]
School of Computing Science, Simon Fraser University, Burnaby, BC, Canada

**Shimin Zhang**[2]
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

**Hieu Nguyen**
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

**Wuyang Chen**
School of Computing Science, Simon Fraser University, Burnaby, BC, Canada

**Vijay Ganesh**
School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA

## Abstract

We present **KissatEvolve**, an LLM-based framework for fully automatic, scalable, and controllable synthesis of SAT solver heuristics. Modern CDCL solvers such as Kissat employ a large ensemble of tightly coupled heuristic components whose joint configuration space is prohibitively large to search exhaustively. Existing LLM-based approaches explore only a narrow slice of this space, confining search to a single solver or a small set of heuristic functions, and optimize exclusively for aggregate PAR-2 without regard for targeted problem subcategories. KissatEvolve is the first framework to address both limitations simultaneously. For **scalability**, KissatEvolve combines a self-adaptive agentic workflow with a fully parallelized generation-and-evaluation pipeline that spans multiple solver families and heuristic functions concurrently, substantially reducing the wall-clock time required to cover the design space while minimizing human intervention. For **controllability**, KissatEvolve maintains a densely annotated memory bank that records both high- and low-performing heuristic variants, each enriched with solver analyses, search statistics, and code-diff traces. Selectively retrieved examples condition subsequent LLM generations toward targeting user-specified subcategories (SAT, UNSAT, easy, or hard problems), enabling fine-grained steering of solver behavior. Using KissatEvolve, we conduct a comprehensive study of 17 heuristic functions across 6 SAT solvers, finding improved heuristics for 3 solvers that reduce PAR-2 by 12.80% on average over their respective baselines. We also find that controlled retrieval reduces PAR-2 by up to 43.76% relative to uncontrolled retrieval.

**2012 ACM Subject Classification** Theory of computation → Automated reasoning

**Keywords and phrases** SAT solving, CDCL solvers, solver heuristics, large language models, program synthesis, evolutionary optimization, automated reasoning

**Supplementary Material** https://github.com/shiminzhang/llm-sat, github.com/m3ru/KissatEvolve, github.com/m3ru/KissatEvolve_RB, github.com/m3ru/KissatEvolve_R1, github.com/m3ru/KissatEvolve_B2

---

[1] Equal contribution. Lead author.
[2] Equal contribution. Listed in alphabetical order by last name.

42
# 1 Motivation and Framework

43 Recent work has begun to automate heuristic generation for SAT solvers (e.g., Kissat [3])
44 by coupling LLMs with evolutionary search loops, enabling the model to propose, compile,
45 evaluate, and refine heuristic code directly [11, 4, 10, 9, 6]. However, these approaches
46 are broadly hindered by two fundamental limitations. The first is **scalability**: existing
47 LLM-based frameworks confine search to a single solver or a narrow slice of the heuristic
48 function space, foregoing the breadth needed to cover the true design landscape, even though
49 no solver dominates across SAT and UNSAT instances [9]. The second is **controllability**:
50 existing frameworks optimize a single aggregate PAR-2 score, collapsing SAT, UNSAT, easy,
51 and hard test instances into an undifferentiated pool, even though competition evaluations
52 report SAT and UNSAT performance independently [1, 2, 7, 5].
53 We answer both questions through **KissatEvolve**, a framework that couples large-
54 scale parallel evolution with retrieval-augmented LLM generation. KissatEvolve operates
55 at the heuristic-function level: it (1) prompts an LLM to propose a natural-language
56 algorithmic modification, (2) translates that proposal into C code, (3) injects the code into
57 the corresponding solver, (4) builds the solver, (5) validates SAT outputs against the CNF
58 and UNSAT outputs via DRAT-trim, and (6) evaluates PAR-2. To enable generation at
59 scale, the pipeline is fully asynchronous and parallelized: parent generation, code synthesis,
60 and build stages are decoupled and connected by queues. This design achieves a 69.41%
61 wall-clock reduction over a sequential baseline (see Appendix B and Appendix C for the
62 comparison to prior LLM-based SAT frameworks).

63
# 2 Memory Bank, Controllability, and Results

64 The memory bank is a pool of past mutation outcomes organized into two FAISS-indexed
65 partitions: a *good* partition for mutants that lowered PAR-2 below their parents and a
66 *bad* partition for mutants that regressed. Each record stores the parent's step-decomposed
67 description, the specific step that was mutated, the resulting mutant description, parent and
68 mutant PAR-2 scores broken down by subcategory (overall, SAT, UNSAT, easy, hard), and an
69 LLM-generated explanatory analysis of the improvement or regression. At the next mutation
70 step, top-$K$ semantically similar successes and failures are retrieved keyed by (parent, step)
71 and injected into the prompt as "what worked" and "what to avoid" exemplars (Appendix B.2).
72 Retrieval is ranked by performance on PAR-2 subcategories, letting researchers trade aggregate
73 PAR-2 cost for sharper subcategory gains without retraining/pipeline changes.
74 As shown in Table 1, KissatEvolve discovers heuristics that improve 3 of 6 targeted solvers,
75 with PAR-2 reductions of 35.4% (Kissat-Cure / `kissat_dynamicsat`) and 1.5% each on the
76 other two, a 12.80% mean reduction over competition baselines. Memory-bank ablations
77 (Appendix C.4) show peak PAR-2 improvements of 80.05 and 118.53 for `kissat_dynamicsat`
78 and `kissat_decide_phase`; controlled retrieval improves PAR-2 in the majority of easy
79 and hard configurations for both heuristics, reducing PAR-2 by up to 43.76% relative to
80 uncontrolled retrieval.

**Table 1** Final solver comparison results. PAR-2 uses a 5000s timeout and 10000s penalty.

| Solver | Heuristic | Best PAR-2 ↓ | Base PAR-2 ↓ |
|---|---|---|---|
| Kissat-Cure | `kissat_dynamicsat` | 2345.57 | 3630.45 |
| Kissat_CoRephase_CoReward | `rephase_conflict` | 2459.43 | 2497.15 |
| AE_kissat2025_MAB | `kissat_bump_score_increment` | 1859.38 | 1887.43 |

## References


1. Tomás Balyo, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*, Helsinki, 2022. Department of Computer Science, University of Helsinki.
2. Tomás Balyo, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, volume B-2023-1 of *Department of Computer Science Series of Publications B*, Helsinki, 2023. Department of Computer Science, University of Helsinki.
3. Armin Biere, Tobias Faller, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT competition 2025. In Cayden Codel, Katalin Fazekas, Marijn J. H. Heule, and Markus Iser, editors, *Proceedings of SAT Competition 2025: Solver and Benchmark Descriptions*, Proceedings of SAT Competitions, page 10, Vienna, 2025.
4. Minyu Chen and Guoqiang Li. DaSAThco: Data-aware SAT heuristics combinations optimization via large language models. *arXiv preprint arXiv:2509.12602*, 2025.
5. Cayden Codel, Katalin Fazekas, Marijn J. H. Heule, and Markus Iser, editors. *Proceedings of SAT Competition 2025: Solver and Benchmark Descriptions*, Proceedings of SAT Competitions, Vienna, 2025. `doi:10.34726/10379`.
6. Hang Ding, Mao Luo, Chu-Min Li, Shunwei Li, Runyao Chen, Caiquan Xiong, and Xinyun Wu. A self-optimizing framework for SAT solvers via population evolution and large language model collaboration. In *Proceedings of SAT Competition 2025: Solver and Benchmark Descriptions*, Proceedings of SAT Competitions, pages 15–17, Vienna, 2025.
7. Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions*, volume B-2024-1 of *Department of Computer Science Report Series B*, Helsinki, 2024. Department of Computer Science, University of Helsinki.
8. Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.
9. 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 *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 6–10. IEEE, 2025.
10. Yiwen Sun, Furong Ye, Zhihan Chen, Ke Wei, and Shaowei Cai. Automatically discovering heuristics in a complex SAT solver with large language models. *arXiv preprint arXiv:2507.22876*, 2025.
11. Yiwen Sun, Furong Ye, Xianyin Zhang, Shiyu Huang, Bingzhen Zhang, Ke Wei, and Shaowei Cai. AutoSAT: Automatically optimize SAT solvers via large language models. *arXiv preprint arXiv:2402.10705*, 2024.
12. Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19724–19731, 2024.


# A Background, Motivations, and Challenges

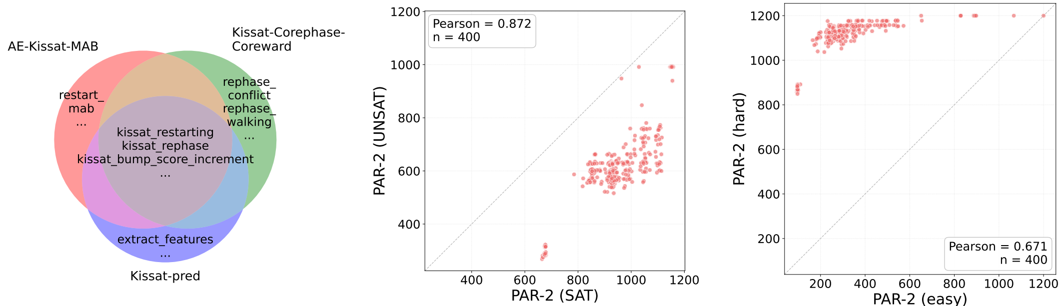
## A.1 Background: Boolean Satisfiability (SAT) Problems, SAT Solvers, and SAT Heuristics

The Boolean Satisfiability Problem (SAT), the first proven NP-complete problem, asks whether a given propositional formula in conjunctive normal form (CNF, a conjunction of clauses, each a disjunction of literals) can be made true by some assignment of its Boolean variables. A SAT solver takes a CNF instance as input and returns either a

satisfying assignment (SAT) or a proof of unsatisfiability (UNSAT). Modern solvers are predominantly based on conflict-driven clause learning (CDCL), with Kissat [3] representing the state of the art. The CDCL workflow interleaves several heuristic-governed components, including branching, restart and rephase policies, and clause reduction, each exposing one or more heuristic functions whose tuning critically affects performance. Solver performance is evaluated via the PAR-2 score: the average runtime over a benchmark set, where instances not solved within a timeout $T$ are penalized at $2T$. Instances are further categorized as SAT or UNSAT based on satisfiability, and as easy or hard based on whether contemporary solvers resolve them well within the timeout.

## A.2 Improving SAT Solvers Demands Exploration over a Large Heuristic Space

The heuristic design space for SAT solvers is vast: even within a single solver family such as Kissat, dozens of variants exist, each instantiating distinct branching, restart, rephase, and clause-management policies. Exhaustive improvement from a single solver is therefore neither principled nor systematic. This is reflected directly in competition practice: top groups simultaneously submit solvers from multiple distinct families precisely because cross-family exploration is necessary [7]. The suboptimal performance of restricting exploration to a single solver is further quantified by the ablation study in DaSAThco [4]: a "Single Best Selection" policy achieves PAR-2 = 1711.1, compared to 1671.0 for the full portfolio that spans multiple heuristic configurations, a 2.4% degradation attributable solely to the narrowed search space.




**(a)** Three Kissat-derived solvers each contribute unique heuristics. No one solver subsumes the others. **(b)** PAR-2 on SAT vs. UNSAT. **(c)** PAR-2 on easy vs. hard.

**Figure 1 Two motivations for KissatEvolve.** (a) Improving SAT solvers requires exploring heuristics across solver families. (b, c) Weak SAT/UNSAT and easy/hard correlations show that heuristic optimization is multi-objective.

As illustrated in Figure 1a, three representative solvers submitted to recent SAT Competitions share a common implementation core, yet each still introduces heuristics absent from the others, and no single solver subsumes the heuristics of the rest. Comprehensive improvement therefore cannot be achieved by refining one solver in isolation: it requires exploring a heuristic space that spans the union of these disjoint regions, motivating frameworks that search across solvers and heuristic families jointly. Despite all of this evidence, no existing LLM-based framework has been designed to jointly evolve heuristics across multiple solver families and function spaces at scale, leaving the vast majority of the design space systematically unexplored.

158 ### A.3 SAT Heuristics Optimization is Multi-Objective

159 In standard SAT Competition evaluation, solver performance on satisfiable and unsatisfiable
160 instances is assessed independently [1, 2, 7, 5]. Rankings on SAT versus UNSAT benchmarks
161 are historically misaligned: a solver that excels on one category frequently underperforms on
162 the other.
163 To make this multi-objective structure concrete, Figure 1(b,c) plots pairwise PAR-2
164 scores across a diverse set of solver configurations generated by our LLM-based framework,
165 partitioning instances into four subcategories: SAT vs. UNSAT, and easy vs. hard (thresholded
166 at PAR-2 = 1000). The relatively low correlation between PAR-2 on each pair of subcategories
167 demonstrates that heuristic choices which improve performance in one regime may fail to
168 transfer to others. Optimizing for SAT does not yield commensurate gains on UNSAT, and
169 vice versa. This Pareto-style tension implies that SAT heuristics optimization is inherently
170 multi-objective, and that controllable, category-aware exploration of the heuristic space
171 is necessary to make targeted progress. Yet no existing LLM-based heuristic discovery
172 framework exposes a steering mechanism for any of these objectives: researchers who require
173 improvements on a specific subcategory are forced to resort to manual solver engineering,
174 precisely the bottleneck that automated heuristic optimization is intended to eliminate.

175 ## B KissatEvolve: Framework Details

176 KissatEvolve operates at a substantially larger scale than prior LLM-based SAT heuristic
177 frameworks, covering 6 Kissat variants and 17 heuristic functions, and is the only system in
178 this comparison to support controllability.
179 KissatEvolve operates at the heuristics (function) level. The core stages of generation
180 are parent generation and mutation (Section B.1.1), connected through a high-throughput
181 execution pipeline (Section B.1.3) and a persistent memory bank (Section B.2). A validation
182 stage checks solving logs and proof status so that invalid solver outputs can be detected
183 and excluded before promotion or reuse. This design allows natural-language ideas, code
184 implementations, and evaluation signals to be linked end-to-end in a reproducible optimization
185 loop.

186 ### B.1 Scalable and Automated Evolution of SAT Heuristics with
187 Diversified Mutation

188 #### B.1.1 Parent Generation with Diversified Mutation

189 **How do we protect against heuristic collapse?**

190 To promote diversity across parents within a single iteration, we linearly vary the sampling
191 temperature from 0.5 to 1.0 over the batch to promote different levels of exploration. We
192 further protect against heuristic collapse with parent monitoring using pairwise cosine
193 similarity (Qwen3-Embedding-0.6B) and an LLM-Judge that flags strategies which are
194 mechanistically equivalent.
195 We iterate the following steps until no PAR-2 improvement is observed:
196 **Step 1: Parent Generation.** The first iteration begins by generating a population of
197 *parent algorithms*: high-level heuristic strategies expressed in *natural language*. We query an
198 LLM with the target heuristic function and its current implementation in Kissat and instruct
199 it to propose a mechanistically distinct strategy for that function. The output specification of

each parent is a JSON containing both an algorithm and a justification for the improvements of that algorithm.

**Step 2: Mutant Generation.** Our contribution is diversified mutations. Instead of asking the LLM to generate loosely related variants of a parent, the key idea is to require the parent description to be written as an explicit sequence of algorithmic steps. The system parses these steps, determines how many mutation opportunities exist, assigns generated variants to specific steps, and instructs the LLM to modify only the assigned step for each mutant.

This design has two advantages. *First*, it improves diversity in a meaningful way: mutants differ along interpretable axes rather than through noisy global edits, while the retrieved exemplars steer each step's modification toward locally-validated trajectories. *Second*, because each mutant is tied to a particular mutated step, the same key supports both storage and retrieval, turning diversified mutation into a self-improving local search whose distribution over edits is shaped by accumulated step-level evidence rather than by a single generic prompt.

**Step 3: Parent Re-selection.** Between iterations of mutations, mutants with the lowest PAR-2 scores are promoted to parents. A subset of the rest are then stored into the densely annotated memory bank (see Section B.2).

### B.1.2 Code Generation

**Code Generation.** Each natural-language algorithm produced in the parent and mutation stages is passed to a separate coder LLM that implements it in C. After the code is ready, our pipeline copies the base solver directory, locates the point in the code and injects the generated algorithm, and then builds and configures the solver so that the binary is ready for evaluation.

**Signal Reference.** To address the codebase awareness concern, we supply the coder with a *signal reference*, which is a markdown sheet enumerating the data structures, types, and solver-internal APIs available to the target function. This reference raised the compilation rate across a team (consisting of a parent algorithm and its mutants) from ∼35% to 90%. This reference was automatically inserted into the coder prompt upon selection of a new target function (see Section C.2) by the pipeline.

### B.1.3 Generating Large-Scale SAT Solvers with High-Performance Parallel Evolution

We implement a pipelined architecture for parallel evolution that achieves an average 69.41% decrease in total wall-clock time compared to sequential execution of the same pipeline stages, allowing for simple scaling of the number of concurrently explored solver candidates. KissatEvolve, like other LLM-based evolution algorithms, experiences the limitations not only of the exploration of the heuristic space, but also by API throughput, compilation time, job submission overhead, and metadata synchronization. We address these bottlenecks through staged queues, semaphores, and dedicated worker pools, preserving the parent-mutant relationships needed for later promotion and experience updates.

**Pipeline Design.** The pipeline consists of three asynchronous stages connected by two queues: (1) algorithm generation pushes parents and mutants into the *code queue*; (2) code-generation workers poll it and enqueue resulting C implementations onto the *build queue*; (3) build-and-submit workers inject the code into a fresh solver copy and dispatch a

SLURM evaluation job array, with PAR-2 results returning asynchronously to drive parent promotion in subsequent iterations.

## B.2 Controllable Heuristics Generation via Densely Annotated Memory Bank

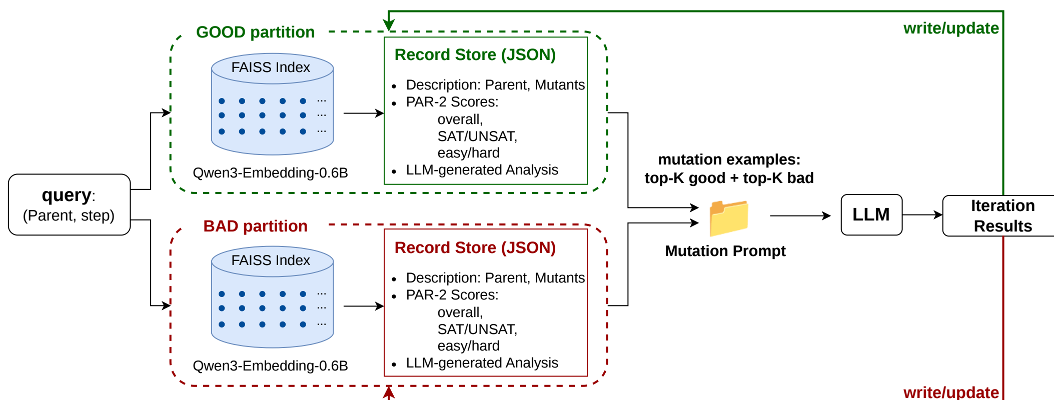



**Figure 2** To enable controllable heuristics generation, we introduce the memory bank for the mutation step. Queries retrieve top-$K$ GOOD and BAD exemplars via FAISS [8] indexing to ground the Mutation Prompt. The memory bank is iteratively updated based on solver iteration results, retaining records ranked by relative PAR-2 change.

We leverage a memory bank so that the algorithm generator learns from prior heuristics, and so that retrieval can be steered toward specific instance subcategories (SAT, UNSAT, easy, hard) when needed. Rather than discarding past attempts or storing only final PAR-2 scores, we adopt a memory bank [12] that stores rich comparative records paired with LLM-generated analyses and exposes them through semantic retrieval.

We illustrate our memory bank design in Figure 2. The pool is organized into two FAISS-indexed partitions, a *good* partition for mutants that lowered PAR-2 below their parents and a *bad* partition for mutants that regressed. Each record stores the parent's step-decomposed description, the specific step that was mutated, the resulting mutant description, parent and mutant PAR-2 scores broken down by subcategory (overall, SAT, UNSAT, hard, easy), and an explanatory analysis.

At the next mutation step, the system retrieves semantically similar past successes and failures keyed by the current (parent, step) and injects them into the mutation prompt as "what worked" and "what to avoid" exemplars, each accompanied by its own short analysis. Search is thus driven not by raw scores alone but by accumulated, code-grounded reasoning about why specific local edits succeeded or failed, making the framework history-aware in a way that compounds across generations.

# C Experiments

## C.1 Settings

Our primary evaluation set is the 400 CNF instances provided by the SAT Competition 2025 [5]. In order to facilitate faster evaluation, we selected a subset of 50 CNF instances

269 and lowered the PAR-2 timing to a quick evaluation standard measured with a 600s timeout
270 per instance and a 1200s penalty on timeout. More details are provided in Section E.1.

271 Evaluations were split between two clusters. Most ran on x86_64 dual-socket AMD
272 EPYC (Zen 5/Turin) nodes (192 cores/768 GB RAM per node, no SMT). Another cluster
273 contains 2 AMD EPYC 7763 (Milan) CPUs with 64 cores each and 512 GB RAM in total.
274 Each solver is built on a single core under an 8 GB memory limit, scheduled via SLURM job
275 arrays.

## 276 C.2 Target Solver Families and Target Functions

277 Kissat [3] is one of the state-of-the-art SAT solvers, and many solvers in the SAT Competi-
278 tion [5] are variants of Kissat. We targeted the best-performing Kissat variants from the
279 SAT Competition 2025 [5], aiming to build a competition-winning solver. Out of the best
280 variants, we selected six final target solvers due to their lower PAR-2 score and their status
281 as a base solver for more unique variants.

282 For AE_kissat2025_MAB [6], the winner from SAT Competition 2025 [5], we generated
283 functions for the restart policy and EVSIDS increment update, two of the most influential
284 heuristics in a CDCL SAT solver. For other solvers, we performed generation on a multitude
285 of heuristics to test KissatEvolve's capability on improving less commonly studied functions.

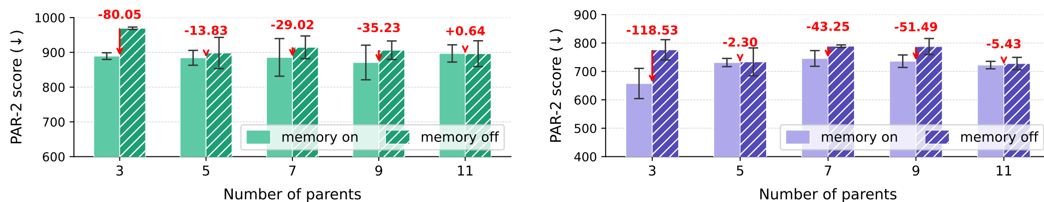
## 286 C.3 Large-Scale Synthesis of SAT Heuristics

287 We begin with pipeline iteration of all generated heuristics on a selected subset of 50
288 problems to efficiently allocate evaluation budget. This screening successfully identifies
289 candidates that surpass their respective baselines: notably, `kissat_dynamicsat` (469.44
290 vs. baseline 531.05), `kissat_bump_score_increment` (475.58 vs. baseline 528.76), and
291 `rephase_conflict` (603.94 vs. baseline 740.72).

292 We then allocate additional computational budget to these promising heuristics and
293 evaluate them on the full benchmark set. The preliminary rankings prove predictive: both
294 `kissat_dynamicsat` (Kissat-Cure) and `kissat_bump_score_increment` (AE_kissat2025_MAB)
295 outperform their baselines on the full 400-instance benchmark, with PAR-2 reductions of
296 35.4% (2345.57 vs. 3630.45) and 1.5% (1859.38 vs. 1887.43), respectively (Table 1 in the
297 main text).

## 298 C.4 Memory Bank Improves PAR-2 Scores

299 To evaluate the effectiveness of the memory bank, we ablate on two heuristic functions
300 from the Kissat-Cure solver: `kissat_dynamicsat` and `kissat_decide_phase`. For each
301 configuration, we run one iteration of diversified mutation with 3, 5, 7, 9, and 11 parents,
302 each generating 3 mutants, and repeat each setting 3 times to reduce variance. Reported
303 PAR-2 values in Figure 3 are the mean across all parents in the generation. The memory
304 bank retrieves at most 3 good and 3 bad examples per query. As shown in Figure 3,
305 enabling the memory bank improves PAR-2 ($\downarrow$) in the majority of configurations, with
306 peak improvements of 80.05 and 118.53 for `kissat_dynamicsat` and `kissat_decide_phase`
307 respectively, demonstrating that conditioning mutation on retrieved experience consistently
308 steers generation toward better-performing heuristics.




**(a)** `kissat_dynamicsat` Heuristics in Kissat-Cure Solver

**(b)** `kissat_decide_phase` Heuristics in Kissat-Cure Solver

**Figure 3** Effect of memory bank and parent count on PAR-2. **Memory on**: memory bank is used to retrieve exemplars during mutation; **Memory off**: no retrieval is used. PAR-2 is evaluated on the 50-instance benchmark subset (Section E.1). Error bars denote 3-run standard deviation.

## C.5 Controllability via Memory Bank

To evaluate controllability, we compare two retrieval strategies: *controlled retrieval*, which re-ranks the top-10 retrieved examples by their PAR-2 score on the target subcategory (easy or hard) before selecting the final 3 good and 3 bad examples, against *uncontrolled retrieval*, which selects examples by semantic similarity alone without subcategory re-ranking. We run both settings once using the same 3, 5, 7, 9, and 11 parent configurations with 3 mutants per parent. As shown in Table 2, controlled retrieval improves PAR-2 in the majority of easy and hard configurations for both heuristics, confirming that subcategory-aware re-ranking effectively steers LLM generation toward instance-specific objectives without requiring changes to the generation pipeline itself.

**Table 2** PAR-2 (↓) of controlled vs. uncontrolled memory bank retrieval on easy and hard instances. **w/**: controlled retrieval; **w/o**: uncontrolled retrieval, examples selected by semantic similarity alone. PAR-2 is evaluated on the 50-instance benchmark subset.

| Heuristics | Split | Parents=3 | | Parents=5 | | Parents=7 | | Parents=9 | | Parents=11 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | w/ | w/o | w/ | w/o | w/ | w/o | w/ | w/o | w/ | w/o |
| `kissat_dynamicsat` | Easy | **280.15** | 461.55 | **256.16** | 455.50 | **559.60** | 562.42 | **244.42** | 417.69 | 443.40 | **439.82** |
| | Hard | **1108.61** | 1138.14 | **1157.07** | 1164.52 | **1142.53** | 1182.69 | **1129.32** | 1157.00 | **1137.16** | 1161.24 |
| `kissat_decide_phase` | Easy | **126.45** | 210.77 | **164.88** | 205.44 | **186.93** | 263.99 | **154.53** | 217.47 | 217.24 | **177.73** |
| | Hard | **954.73** | 1022.19 | **998.78** | 1014.38 | **1001.76** | 1075.69 | **949.52** | 1053.62 | **961.20** | 1018.26 |

# D Limitations

Our most salient limitation in KissatEvolve is the evaluation noise. Cluster heterogeneity and load dictate the number of instances solved, which in turn affects PAR-2 score. To mitigate this, we scheduled evaluation jobs for our most promising solvers around 12:00 AM PDT to minimize cluster load. We also maintained a clear record of how many instances each generated solver finished in order to judge whether two PAR-2 scores are comparable.

# E Implementation Details

For every candidate included in any reported table, we independently validate SAT outputs by checking the produced assignment against the CNF and validate UNSAT outputs using DRAT-trim on the emitted proof. Candidates that fail validation on any instance are assigned timeout penalty and are never promoted or inserted into the memory bank.

## E.1 Stratified Benchmark Subset Construction

To enable rapid iteration during heuristic search, we constructed a stratified 50-instance subset of the SAT Competition 2025 benchmark [5]. We first ran the unmodified base solver on the full 400-instance set with a 5000s wall-clock timeout to obtain a per-instance solving time, then bucketed instances by runtime into three difficulty tiers: Easy (<1000s), Hard (1000–5000s), and Timeout. For quick evaluation, we sampled 50 instances proportionally using a fixed random seed (42) for reproducibility. The resulting subset preserves the full benchmark's difficulty profile while substantially reducing per-iteration evaluation cost.

# F Best-Generated Heuristic: Code Example

Generated code

```
void kissat_bump_score_increment (kissat *solver) {
  // Step 1: Calculate the base EVSIDS multiplier factor M = 1.0 / (1.0 - decay)
  const double decay_per_mille = GET_OPTION (decay);
  const double decay = decay_per_mille * 1e-3;
  const double M = 1.0 / (1.0 - decay);

  // Step 2: Retrieve the short-term (fast) and long-term (slow) EMAs of the LBD
  // In Kissat, these are stored as fast_glue and slow_glue in the averages
      struct.
  const double glue_fast = solver->averages[solver->stable].fast_glue.value;
  const double glue_slow = solver->averages[solver->stable].slow_glue.value;

  // Step 3: Calculate the Learning Efficiency Ratio rho = glue_fast / glue_slow
  const double rho = (glue_slow > 0.0) ? (glue_fast / glue_slow) : 1.0;

  // Step 4: Compute a Momentum Adjuster A = 1.0 + clamp(0.1 * (rho - 1.0),
      -0.02, 0.02)
  double diff = 0.1 * (rho - 1.0);
  if (diff < -0.02)
    diff = -0.02;
  else if (diff > 0.02)
    diff = 0.02;
  const double A = 1.0 + diff;

  // Step 5: Update the global score increment using an asymmetric scaling
      formula
  const double adj = (A > 1.0) ? (A * A) : A;
  solver->scinc *= (M * adj);

  LOG ("new score increment %g (rho %g, A %g, adj %g)", solver->scinc, rho, A,
      adj);

  // Step 6: Momentum-Aware Rescale
  if (solver->scinc > MAX_SCORE) {
    if (rho > 1.0) {
      // Apply a 'Deep Compression' by multiplying all variable scores by 1e-160
      // We achieve this by setting scinc to 1e160 so kissat_rescale_scores uses
          factor 1e-160
      solver->scinc = 1e160;
      kissat_rescale_scores (solver);
      // Reset solver->scinc to 1.0 as per instructions
      solver->scinc = 1.0;
    } else {
      // Perform the standard kissat_rescale_scores (factor ~1e-150)
      kissat_rescale_scores (solver);
    }
  }
}
```

A step-by-step technique analysis of the generated function (along with the baseline code for comparison) is available in our supplementary materials.