AutoSAT: Automatically Optimize SAT Solvers via Large Language Models

YIWEN SUN, School of Data Science, Fudan University, China
FURONG YE, Key Laboratory of System Software, Institute of Software, Chinese Academy of Sciences, China
XIANYIN ZHANG, School of Data Science, Fudan University, China
SHIYU HUANG, 4Paradigm Inc., China
BINGZHEN ZHANG, Department of Industrial Engineering, Tsinghua University, China
KE WEI*, School of Data Science, Fudan University, China
SHAOWEI CAI*, Key Laboratory of System Software, Institute of Software, Chinese Academy of Sciences, China

Background: Conflict-Driven Clause Learning (CDCL) is a dominant framework for solving the Satisfiability problem (SAT). Modern CDCL solvers rely heavily on various heuristics, which significantly influence their performance. Established solvers such as MiniSat and Kissat typically incorporate multiple heuristics and therefore require substantial manual effort and domain expertise for fine-tuning in practice.
Objectives: The emergence of Large Language Models (LLMs) offers a promising opportunity to automate SAT solver optimization. However, generating a complete CDCL solver from scratch using LLMs is impractical due to the complexity and large context volume of modern SAT solvers. To address this challenge, we propose AutoSAT, a framework that automatically optimizes heuristics within a CDCL solver, EasySAT. Our goal is to leverage LLMs to discover effective heuristic designs beyond conventional parameter tuning.
Methods: Unlike traditional automated algorithm design approaches that mainly focus on hyperparameter tuning and operator selection, AutoSAT can generate new efficient heuristics for CDCL solvers. In this first attempt to leverage LLMs for SAT solver optimization, we integrate several search strategies, including the greedy hill climber and the $(1 + 1)$ Evolutionary Algorithm, to guide LLMs in searching for better heuristics.
Results: Experimental results demonstrate that LLMs can consistently enhance the performance of CDCL solvers. Empirically, AutoSAT outperforms MiniSat and its parameter-tuning variants on 12 of the 19 test datasets, and even surpasses the state-of-the-art hybrid solver Kissat and its parameter-tuning variants on 4 datasets.
Conclusions: These results suggest that LLMs can serve as effective agents for automatically improving SAT solver heuristics. AutoSAT provides an initial step toward automated heuristic discovery for CDCL solvers, showing the potential of LLM-based methods to complement and reduce the manual expertise traditionally required in SAT solver design.

JAIR Track: Constraint Programming and Machine Learning

*Corresponding authors

Authors' Contact Information: Yiwen Sun, ORCID: 0000-0002-8681-4165, ywsun22@m.fudan.edu.cn, School of Data Science, Fudan University, Shanghai, China; Furong Ye, ORCID: 0000-0002-8707-4189, f.ye@ios.ac.cn, Key Laboratory of System Software, Institute of Software, Chinese Academy of Sciences, Beijing, China; Xianyin Zhang, ORCID: 0009-0006-7675-9429, xianyinzhang22@m.fudan.edu.cn, School of Data Science, Fudan University, Shanghai, China; Shiyu Huang, ORCID: 0000-0003-0500-0141, huangsy1314@163.com, 4Paradigm Inc., Beijing, China; Bingzhen Zhang, ORCID: 0009-0009-2527-3092, zhangbz23@mails.tsinghua.edu.cn, Department of Industrial Engineering, Tsinghua University, Beijing, China; Ke Wei, ORCID: 0000-0003-1222-3044, kewei@fudan.edu.cn, School of Data Science, Fudan University, Shanghai, China; Shaowei Cai, ORCID: 0000-0003-1730-6922, caisw@ios.ac.cn, Key Laboratory of System Software, Institute of Software, Chinese Academy of Sciences, Beijing, China.

DOI: 10.1613/jair.1.20499

JAIR Associate Editor: Tias Guns

JAIR Reference Format:
Yiwen Sun, Furong Ye, Xianyin Zhang, Shiyu Huang, Bingzhen Zhang, Ke Wei, and Shaowei Cai. 2026. AutoSAT: Automatically Optimize SAT Solvers via Large Language Models. *Journal of Artificial Intelligence Research* 86, Article 29 (July 2026), 50 pages. DOI: 10.1613/jair.1.20499

1 Introduction

As the first proven NP-complete problem (Cook 1971), the Satisfiability problem (SAT) is fundamental to numerous domains in computer science. Meanwhile, SAT solvers have shown significant impact in a wide range of applications, including software verification, program analysis, electronic design automation. Conflict-Driven Clause Learning (CDCL), introduced in (Silva and Sakallah 1996), is a dominant paradigm for SAT (Audemard and Simon 2009; Biere 2019; Fleury and Heisinger 2020; Kochemazov et al. 2019; Mahajan et al. 2004; Moskewicz et al. 2001), underpinning almost all state-of-the-art (SOTA) SAT solvers. An alternative approach is local search, which is an incomplete method that prioritizes searching for satisfiable assignments and can be competitive on random and hard combinatorial instances (Balint and Fröhlich 2010; Cai, Luo, et al. 2015; Hoos and Stützle 2000). The hybrid solvers combining CDCL and local search (Audemard, Lagniez, et al. 2010; Biere 2019; Cai and X. Zhang 2021; Fleury and Heisinger 2020) have recently achieved SOTA results in SAT Competitions. This work focuses on CDCL solvers, whose continued progress primarily relies on the development of novel heuristics, including branching heuristics, restart heuristics and clause reduction heuristics. These heuristics are typically designed based on expert intuition and refined through trial-and-error approaches (Fleury and Heisinger 2020; Kullmann 2021; Souza and Ritt 2018).

Designing heuristics often requires significant time and human effort. Moreover, the ongoing development of various methods makes it difficult to select the most suitable one for a specific problem instance. To address this issue, there has been a line of research that focuses on selecting a good configuration (e.g., the settings of parameters or algorithmic components) from a predefined configuration space (Adriaensen et al. 2022; Hoos, Hutter, et al. 2021; Kerschke et al. 2019; Luo et al. 2020; Speck et al. 2021). However, this approach relies on manual design of the configuration space, and the achievable performance is therefore limited by that space. Fortunately, Large Language Models (LLMs) (Achiam et al. 2023; Brown et al. 2020; Naveed et al. 2023) provide a promising way to overcome this limitation as they can generate new code automatically. Recent research has experimentally demonstrated the ability of LLMs for programming tasks (Chen et al. 2023; Josifoski et al. 2023; R. Li et al. 2023; Nam et al. 2024; Wu et al. 2023; H. Yang et al. 2023). For example, FunSearch (Romera-Paredes et al. 2024) uses LLMs to generate solutions in the form of programs for the cap set and knapsack problems and iteratively evolves them toward better solutions. ReEvo and EoH (F. Liu et al. 2024; H. Ye et al. 2024) apply LLMs to generate heuristics for problems such as online bin packing, traveling salesman, and flow shop scheduling. LLaMEA combines evolutionary algorithms with LLMs to search metaheuristics automatically for fundamental continuous optimization problems (Stein and T. Bäck 2024).

Motivated by the programming capabilities of LLMs, this work investigates whether they can assist in optimizing heuristics for CDCL solvers. Specifically, we investigate the following two questions:

- Can LLMs generate a more efficient implementation of a CDCL solver than existing human-designed ones?
- Given a SAT solver, can LLMs discover better heuristics and improve the implementation?

In response to the first question, although LLMs can generate code within the basic CDCL framework, it is observed that the resulting code tends to be too simplified to be competitive primarily due to the limited context length, see Appendix C.1. Thus, the answer to the first question is negative, and LLMs cannot develop a complete CDCL solver from scratch that can match the performance of human-designed implementations.

Therefore, we shift our focus to the second question and adopt an alternative approach: we supply LLMs with the source code of an existing solver and leverage them to modify the code directly. Our findings indicate that LLMs can identify many potential directions for modifying existing code. Thus, we propose leveraging LLMs to modify heuristic functions, given the original solver code as the context. The main contributions of this paper are summarized as follows:

- **AutoSAT: A competitive and extensible SAT solver optimization framework.** We introduce AutoSAT, an open-source framework that autonomously optimizes embedded heuristics in a CDCL solver known as EasySAT (Zhihan Chen 2022) for given datasets. Experimental results show that AutoSAT substantially enhances a simple CDCL solver EasySAT (with hundreds of lines of code), and enables it to achieve competitive performance compared to the modern solver MiniSat which has been widely used in industry. Moreover, solvers produced by AutoSAT outperform the SOTA hybrid solver Kissat on several datasets. In addition, the modular design of AutoSAT facilitates the integration of additional techniques, making it possible for further performance improvements in the future.
- **A novel scheme for LLM-based optimization in complex scenarios.** Unlike existing approaches that rely on LLMs to generate entire programs from scratch, we leverage them to optimize solvers over a fine-grained search space. This offers a more practical and scalable pathway for applying LLMs to complex and large-scale real-world programming tasks.
- **Comprehensive analysis of LLM-based SAT solvers.** Based on the fine-grained space defined for CDCL solvers, we empirically investigate (1) how LLMs iteratively optimize the SAT solver and (2) how the generated heuristics contribute to performance gains. These discussions provide valuable insights for future research on LLM-driven solver optimization, such as the effective budget allocation for LLM queries.

2 Preliminaries and Related Works

2.1 Preliminary Definition of SAT

Let $V = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean variables. A *literal* is either a variable x or its negation $\neg x$. A *clause* is a disjunction of literals. A *conjunctive normal form* (CNF) formula $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ is a conjunction of clauses. For simplicity, we assume all clauses are non-tautological, in other words, no variable x occurs positively ($x \in C$) and negatively ($\neg x \in C$) in the same clause.

A (partial) mapping $\alpha : V \to \{0, 1\}$ is called an *assignment*. If α maps all variables to a boolean value, it is termed a *complete* assignment; otherwise, it is referred to as a *partial* assignment. The value of a variable x under an assignment α is denoted as $\alpha[x]$. An assignment α satisfies a clause if at least one literal evaluates to true under α, and satisfies a CNF formula if it satisfies all its clauses. A CNF formula F is satisfiable if there exists at least one satisfying assignment. The empty clause $\perp$ is always unsatisfiable and represents a *conflict*. SAT is the problem of determining whether a given CNF formula is satisfiable.

2.2 CDCL Solver

A CDCL solver follows a propagate-and-learn loop, interleaved with a decision guessing phase. While the core propagate-and-learn mechanisms are largely standardized across different solvers, the policies for making decisions and scheduling restarts can vary significantly and have a major impact on performance. These policies are typically hand-crafted and lack formal guarantees, hence the term heuristics. Over the years, numerous heuristic techniques have been developed to enhance the efficiency of CDCL solvers, and their design remains an active research area in SAT solving.

Branching heuristics play a crucial role in the performance of CDCL solvers. For example, Variable State Independent Decaying Sum (VSIDS) (Biere, Heule, et al. 2009) is a family of branching heuristics that selects the most promising variable in the Make Decision phase. Another important branching heuristic is Learning-Rate

Branching (LRB) (Liang, Ganesh, et al. 2016), which formulates branching as an optimization problem and selects the variable that maximizes a metric known as learning rate.

Restart heuristics are also essential for CDCL solvers. A restart abandons the current search path and backtrack to a specific decision level while retaining the learned clauses. Static Luby restart schedules were historically popular due to the optimality (in expectation) for unknown heavy-tail distributions (Luby et al. 1993). Fast restart (Ramos et al. 2011) later gained popularity by restarting at short, dynamically-adjusted intervals to exploit lucky runs and escape unpromising branches. Subsequently, most implementations switched to Glucose-style restarts (Audemard and Simon 2012), which became a de facto standard for success in SAT Competitions. For an extensive overview of these techniques, see the survey presented in (Biere and Fröhlich 2015).

Reduction heuristics are essential for managing learned clauses in CDCL solvers. The accumulation of learned clauses would slow down propagation and degrade memory efficiency. Reduce heuristics periodically deletes a subset of clauses to maintain a manageable clause database while retaining the most critical ones for pruning the search space. Classical approaches such as the Glucose reduction policy (Audemard and Simon 2012) prioritize clauses with low Literal Block Distance (LBD), reflecting their structural importance. Other work explores more adaptive strategies, for example dynamically adjusting deletion thresholds based on clause activity, age, or size (Audemard and Simon 2009). The core challenge lies in balancing aggressive removal with the risk of discarding valuable information, a trade-off that often determines the long-term efficiency and scalability of the solver.

Rephase is another heuristic integrated into CDCL solvers. It enables the solver to explore different regions of the search space by resetting or adjusting the current partial assignment. PrecoSAT and PicoSAT (Biere 2010) utilize a Jeroslow-Wang score (Jeroslow and J. Wang 1990) to adjust the saved phases either on all clauses or only on the irredundant ones at regular intervals, following a Luby sequence. StrangeNight (Soos 2013) employs a probabilistic value-flipping strategy, where the probability depends on the current decision depth, aiming to mitigate the heavy-tail phenomenon. A comprehensive evaluation of rephasing heuristics has been conducted in the context of the SAT solver Riss (Balint, Belov, et al. 2015).

2.3 LLM-based Optimization

Leveraging Large Language Models (LLMs) to optimize algorithms has emerged as a growing research area, particularly after DeepMind's success with FunSearch (Romera-Paredes et al. 2024), where LLMs are used to solve complex combinatorial problems such as bin-packing problems. Although FunSearch demands substantial computational resources that limit its general applicability, it clearly demonstrates the potential of LLMs in tackling hard optimization tasks. This has spurred interest in LLM-based approaches across diverse domains. For example, in (F. Liu et al. 2024), they utilize LLMs to solve traveling salesman problems and scheduling problems, and follow-up studies have adopted similar techniques for evolving the acquisition functions in Bayesian Optimization (Yao et al. 2024) and for adversarial attacks (Guo et al. 2024). Additionally, LLaMEA (Stein and T. Bäck 2024) studies the LLMs-based approach for black-box continuous optimization, with performance evaluated on the established BBOB benchmark (Hansen and Ros 2010).

Most of the current research utilizes evolutionary algorithms in the search for better algorithm designs. The pioneering FunSearch work applies the islands model (a parallel genetic algorithm) which requires numerous LLM requests per trial. More recent studies show that simpler approaches, such as the $(1 + 1)$ Evolutionary Algorithm (EA), can yield promising results within fewer requests. A similar trend was also observed in EoH (F. Liu et al. 2024), which employed population-based genetic algorithms. A comprehensive benchmarking study in (R. Zhang et al. 2024) compared various evolutionary search policies for LLM-based optimization and revealed distinct behaviors and efficiencies among different approaches.

2.4 The (1+1) EA Algorithm in Evolutionary Computation

Evolutionary computation plays an important role in black-box optimization and, as noted earlier, has also influenced LLM-based optimization. The (1 + 1) EA (Droste et al. 2002) algorithm is a simple yet popular evolutionary algorithm. It can be viewed as a special case of the more general $(\mu + \lambda)$ EAs, where λ is the number of new offspring solutions and μ is the number of solutions that are selected for the next generation. In the (1 + 1) EA, both μ and λ are set to one, meaning that each generation produces a single offspring solely via mutation. This design offers several advantages: it is conceptually simple, requires no population management, and allows for tight theoretical analysis, which has led to extensive benchmarking and rigorous runtime studies. Furthermore, the algorithm's strong exploitation capability and low computational overhead make it particularly appealing for evolving SAT solver heuristics. Motivated by these properties, this paper adopts the (1 + 1) EA as the search strategy for optimizing SAT solvers. For more comprehensive details of EAs, we refer readers to the recent surveys (T. H. Bäck et al. 2023; B. Doerr and Neumann 2019).

2.5 Machine Learning for Designing Heuristics in SAT Solvers

Recent advances in SAT solver design reveal a clear trend: machine learning is increasingly used to select or design heuristics. Heuristics that were once handcrafted by experts are now frequently augmented with multi-armed bandit (MAB) strategies or graph neural networks (GNNs). These approaches leverage both statistical features of the solver's runtime behavior and structural properties of CNF formulas to enable more adaptive, instance-specific decisions.

For branching heuristics, MapleCOMSPS (Liang, Oh, Ganesh, et al. 2016) introduced the learning rate branching (LRB) heuristic, which measures how quickly each variable contributes to learning clauses. In MapleCOMSPS, variable selection is formulated as a multi-armed bandit (MAB) problem, using an exponential recency-weighted average method. In contrast, Kissat_MAB (Cherif et al. 2021) leverages MAB in a different manner, dynamically switching between VSIDS and conflict history-based branching (CHB) via an upper confidence bound policy, where a reward function estimates conflict quality. Both solvers achieved top rankings in SAT Competitions at the time and inspired numerous follow-up works. Following NeuroSAT (Selsam, Lamm, et al. 2019), which firstly formulates CNF as a GNN to process, neural networks have been utilized to design heuristics within SAT solvers. For instance, NeuroCore (Selsam and Bjørner 2019) periodically replaces the activity scores of VSIDS in MiniSat with the output of neural networks – a technique termed periodic refocusing.

For restart heuristics, a new policy called machine learning-based restart (MLR) (Liang, Oh, Mathew, et al. 2018) is designed to trigger a restart when the predicted LBD of the next learned clause exceeds a certain threshold. MLR uses the LBDs of the last three learned clauses and their products as features to fit a linear function that predicts the LBD of the next learned clause. On SAT Competition benchmarks, MLR achieves better performance than the Luby restart policy.

For reduce heuristics, Vaezipoor et al. (Vaezipoor et al. 2020) formulate this task as a reinforcement learning problem and optimize a policy that outputs an LBD threshold as an action, deleting all clauses above that threshold. NeuroSelect (H. Liu et al. 2024) reframes clause management in CDCL solvers as instance-wise policy selection rather than per-clause scoring. The authors first diversify candidate policies by augmenting Kissat's default deletion rule with a complementary metric based on variable propagation frequency. To choose between the default policy and the frequency-guided policy on a given instance, they train a hybrid graph transformer that combines local message passing with global linear attention over variable nodes, enabling it to capture long-range variable interactions at nearly-linear cost in terms of graph size.

For phasing heuristics, NeuroBack (W. Wang et al. 2021) trains a lightweight graph transformer to predict per-variable phases by approximating backbone information from a CNF's bipartite graph representation. The predicted phases are injected as the initialization for the CDCL solver's phase selection heuristic, avoiding the

Algorithm 1: CDCL Framework

Input: A CNF F of SAT instance;
Initialization: decision level $d \leftarrow 0$, current assignment of variables $\mathcal{X} \leftarrow \emptyset$;
while ***True*** **do**
  $\mathcal{X} \leftarrow$ **Unit Propagation**$(F, \mathcal{X})$;
  if *Conflicts are detected in* $\mathcal{X}$ **then**
    if $d == 0$ **then**
      return UNSAT;
    else
      $C_{conflict}, d_{backtrack} \leftarrow$ **Analyze Conflict**$(F, \mathcal{X})$;
      $C_{learned} \leftarrow$ **Learn Clause**$(C_{conflict}, \mathcal{X})$;
      $F \leftarrow F \wedge C_{learned}$;
      $\mathcal{X} \leftarrow$ **Backtrack**$(\mathcal{X}, d_{backtrack})$;
      $d \leftarrow d_{backtrack}$;
  else
    if *All variables are assigned with a value* **then**
      return SAT;
    else
      $d \leftarrow d + 1$;
      $\mathcal{X} \leftarrow$ **Make Decision**$(\mathcal{X})$;

need for frequent online queries at runtime. This approach is integrated into Kissat and trained on a 120k-instance dataset, improving the number of solved instances on SAT Competition 2022 and 2023, thereby demonstrating its practical effectiveness on large instances.

3 AutoSAT

In this section, we introduce AutoSAT, an LLM-based optimization framework for CDCL solvers. AutoSAT aims to enhance SAT solver performance by leveraging LLMs to iteratively search for more effective heuristics.

We first describe the architecture of the CDCL solver and outline the corresponding search space. Next, we examine two search strategies, the greedy hill climber (GHC) and the (1 + 1) EA, and present a comparative analysis. Lastly, we illustrate the prompt engineering techniques employed in AutoSAT.

3.1 Fine-Grained Search Space

Although previous studies mentioned in Section 2.3 have successfully addressed similar optimization problems in other domains, none have considered SAT, which requires high computational costs. SAT solvers typically employ advanced programming techniques to achieve results within minimal CPU time. Over decades of development, SOTA SAT solvers such as Kissat have integrated numerous heuristics. However, in our preliminary experiments, using LLMs to generate a SAT solver from scratch following the EoH mechanism only produces a basic algorithm (see Appendix C.2). Even after several iterations, the generated solver tended to incorporate only a single heuristic module for improvement. Moreover, guiding LLMs to design competitive data structures remains particularly challenging, especially when compared to the highly optimized implementations of SOTA SAT solvers that have been refined over decades.

To address these challenges, we employ a predefined CDCL framework as presented in Alg. 1. A CDCL algorithm typically starts with an empty set of partial assignments (line 2). Unit Propagation (UP), also known as Boolean Constraint Propagation, assigns values only to those variables that can make a clause satisfied (i.e., all other literals in this clause are false). This operation repeats until no more UP is possible (line 4). If no conflicts are detected in $\mathcal{X}$ during the Decision Detection phase, the algorithm selects a variable and assign a value to it (line 19). This Make Decision step usually follows a heuristic. When a conflict is detected (line 5), Analyze Conflict will identify the conflicted clauses (line 9), and a newly learned clause will be learned based on the cause (i.e., the current partial assignment) of conflicts (line 10). Then the algorithm backtracks to an earlier decision level (lines 12-13). Once all variables are assigned and no conflict is detected, the algorithm obtains a satisfying assignment (line 16). On the other hand, detecting a conflict at the decision level 0 indicates that the given CNF is unsatisfiable (line 7).

Modern SAT solvers are mostly built on the CDCL framework, and various heuristics have been continuously proposed to enhance their performance. For instance, *reduce* heuristics work on the tracks of learned clauses (referring to line 10), control the size of the tracking list, and determine which learned clauses to remove. In addition, *bump var heuristics* are often integrated with Analyze Conflict (line 9) and influence variable selection in the Make Decision function (line 19). While the order of choosing variables can determine the search path of branching, *rephase* heuristics can control the polarity in variables to be selected. Additionally, *restart* heuristics may abandon the current search path, allowing algorithms to explore possibly easier search regions.

In this paper, we use AutoSAT to improve an open-source SAT solver EasySAT (Zhihan Chen 2022), which implements a basic CDCL framework and has roughly five hundred lines of C++ code. Within EasySAT, we define nine independently implemented functions: **restart**, which manages restart heuristics, **restart condition update**, which adjusts restart conditions, **restart condition**, which determines when to execute restart, **reduce**, which handles reduce heuristics, **reduce condition**, which determines when to reduce, **rephase**, which manages rephase heuristics, **rephase condition**, which determines when to rephase, and two functions **bump var** and **bump var heuristic** that govern the order of variables being selected during conflict analysis. AutoSAT maps from a set of heuristics to a solver, $H : \{h_1, h_2, \ldots, h_9\} \mapsto A$, where each h_i represents the heuristics for a function, and the nine functions together form a customized CDCL Solver A. The candidate heuristics for each h_i are generated by the LLMs.

3.2 Search Strategies

Optimization Problem Based on the search space defined above, the task of optimizing a CDCL solver can be formulated as a minimization problem. Given the search space $\boldsymbol{H} = \{(h_1, \ldots, h_9) \mid h_i \in H_i,\ i = 1, \cdots, 9\}$, where H_i denotes the domain (i.e., a set of function candidates) for the i-th heuristic, and given a set of SAT instances P, the goal is to find a configuration $\boldsymbol{h}^*$ such that

$$\boldsymbol{h}^* \in \arg\min_{\boldsymbol{h} \in \boldsymbol{H}} f_P(\boldsymbol{h}). \tag{1}$$

Each $\boldsymbol{h} \in \boldsymbol{H}$, an specific combination of the 9 heuristics, uniquely determines a solver A, and the objective function $f_P(\boldsymbol{h})$ is evaluated by the PAR-2 value of the solver A (constructed from $\boldsymbol{h}$) across the instance set P. For the ease of presentation, we denote $f_P(\boldsymbol{h}) = f_P(A)$.

Note that characterizing the size of the candidate set for each heuristic function h_i remains challenging. It is worth emphasizing that H_i is not predefined in this work, which distinguishes our approach from prior work on automatic solver configuration that operates within a predefined configuration space.

Algorithms We adopt the *greedy hill climber (GHC)* and $(1 + 1)$ *Evolutionary Algorithm (EA)* as the search strategies within AutoSAT for optimizing CDCL solvers. The GHC modifies the heuristic functions sequentially, from the first to the last. It can efficiently achieve immediate improvement during the search, but at the same

Algorithm 2: AutoSAT Framework with the Greedy Hill Climber (AutoSAT_GHC)

1 **Input:** A group of SAT instances P, a CDCL framework formed by nine independent functions $\{h_1, h_2, \ldots, h_9\}$, a prompt template based on the current best A, indexes $M \leftarrow \emptyset$ of functions to be modified, and $\mathcal{B}$ represents the number of requests for new heuristics from LLMs;
2 **Initialization:** Assign a predefined $h \leftarrow \{h_1, h_2, \ldots, h_9\}$, and initialize a solver A based on h, $f^* \leftarrow$ evaluate the performance of A;
3 **for** $i \in \{0, \ldots, \mathcal{B} - 1\}$ **do**
4 Get function index $m \leftarrow i \mod 9$;
5 Form a new prompt based on A and m, and request new heuristics functions $\{h'_m\}$ from LLMs;
6 $h' \leftarrow$ replacing the functions in h by the corresponding functions in $\{h'_m\}$;
7 Obtain a new solver A' based on h', and $f_P(A') \leftarrow$ evaluate the performance of A';
8 **if** $f_P(A') \leq f^*$ **then**
9 $A \leftarrow A'$ and $h \leftarrow h'$;
10 $f^* \leftarrow f_P(A')$;

time can easily get stuck in local optima due to its strict improvement criterion and lack of diversity mechanisms. The (1+1) EA is a classic evolutionary algorithm that maintains a single parent solution and generates offspring by modifying one or more randomly selected heuristic functions. Compared to GHC, it offers advantages in exploration. Both algorithms are well-suited to our optimization scenario, and many fundamental studies have been conducted for their experimental and theoretical properties (C. Doerr, F. Ye, Horesh, et al. 2020). We therefore adopt both approaches, providing a baseline for future comparisons.

We begin with the presentation of AutoSAT embedded with the GHC in Alg. 2. AutoSAT starts with a solver whose nine heuristics functions follow the default settings in EasySAT (Zhihan Chen 2022)[1]. In each iteration, AutoSAT modifies a selected heuristic function to generate a new solver, and the nine heuristic functions are processed in order the first one to the last one (line 4). After selecting which heuristic function to modify, AutoSAT requests LLMs to provide alternative implementations for the chosen heuristic function (lines 5-6). In each iteration, based on the LLM-generated response, AutoSAT generates and evaluates the new solver A' (lines 7). If A' outperforms the current best solver A, AutoSAT greedily updates the corresponding heuristics and incorporates the best implementation found so far (lines 8-10).

Although the GHC can achieve immediate improvement when optimizing the SAT solvers, it can be easily trapped in a local optimum without any diversity mechanisms. In addition, modifying heuristic functions one by one is effective under the assumption that different heuristic functions contribute to the performance of the SAT solvers independently, which is unlikely to hold in practice. For example, three heuristics functions restart condition, restart condition update, and restart cannot work independently. Therefore, we also evaluate the performance of $(1 + 1)$ EA for AutoSAT.

In Alg. 3, AutoSAT again starts with an initial solver that follows the default settings of EasySAT (Zhihan Chen 2022). It then requests LLMs to provide alternative implementations for ℓ distinct heuristic functions (lines 4-6). Instead of modifying a deterministically chosen heuristic function (line 4 in Algorithm 2), the $(1 + 1)$ EA approach selects ℓ distinct heuristic functions to mutate (line 5), where ℓ follows a static binomial distribution (line 4). This allows AutoSAT to generate a new solver A' by modifying one or multiple heuristic functions (lines 7-8). If A'

[1]We use EasySAT instead of MiniSat or Kissat due to its light volume. More precisely, EasySAT requires only $5k$ tokens when input into LLMs, in contrast to MiniSat's $25k$ tokens and Kissat's $250k$ tokens, making it more manageable for LLMs to understand and revise. Furthermore, Kissat lacks modularity in its codebase, resulting in a tangled structure that poses significant challenges for LLMs to modify.

Algorithm 3: AutoSAT Framework with (1 + 1) EA (AutoSAT_EA)

1 **Input:** A group of SAT instances P, a CDCL framework formed by nine independent functions $\{h_1, h_2, \ldots, h_9\}$, a prompt template based on the current best A and indexes $M \leftarrow \emptyset$ of functions to be modified, and $\mathcal{B}$ represents the number of requests for new heuristics from LLMs;
2 **Initialization:** Assign a predefined $h \leftarrow \{h_1, h_2, \ldots, h_9\}$, and initialize a solver A based on h, $f^* \leftarrow$ evaluate the performance of A;
3 **for** $i \in \{0, \ldots, \mathcal{B} - 1\}$ **do**
4 Sample a value $\ell \sim \text{Bin}(9, \frac{1}{9})$;
5 chosen ℓ distinct values $M \leftarrow \{m_1, \ldots, m_\ell\}$ from $\{1, \ldots, 9\}$ uniformly at random;
6 Form a new prompt based on A and M, and request new heuristics functions $\{h'_{m_1}, \ldots, h'_{m_l}\}$ from LLMs;
7 $h' \leftarrow$ replacing the functions in h by the corresponding functions in $\{h'_{m_1}, \ldots, h'_{m_l}\}$;
8 Obtain a new solver A' based on h', and $f_P(A') \leftarrow$ evaluate the performance of A';
9 **if** $f_P(A') \leq f^*$ **then**
10 $A \leftarrow A'$ and $h \leftarrow h'$;
11 $f^* \leftarrow f_P(A')$;

outperforms the current best solver A, AutoSAT greedily updates the prompts and then incorporates the best implementation found so far (lines 9-11).

It is worth noting that the fine-grained search space of AutoSAT allows us to leverage existing research in discrete optimization to design the search strategies. Recalling the optimization problem defined earlier, if LLMs can suggest the optimal candidate for each function h_i, then the problem reduces to a pseudo-Boolean optimization problem since the domain of each heuristic function contains only (0) the default setting and (1) the optimal candidate. Based on this assumption and considering that the performance of a heuristic function $\boldsymbol{h} \in \boldsymbol{H}$ may depend on interactions among different functions, searching for the optimal AutoSAT solver can be formulated as a sequential decision problem. Drawing from theoretical runtime analysis results (Böttcher et al. 2010), we set our budget $B \approx 0.6n^2, n = 9$ (where n means the number of dimensions that can be modified in (1+1)EA algorithm), following the established benchmark and theoretical insights on LeadingOnes (C. Doerr, F. Ye, Rijn, et al. 2018; F. Ye et al. 2020). This is a problem where the contribution of each variable is conditional on all preceding variables, and thus offers an analogue to the sequential dependencies present in our setting.

3.3 Prompt Engineering

Prompt engineering is essential for effective use of LLMs. It ensures that LLMs operate within the intended context and can build upon previously generated outputs, e.g., new heuristic implementations produced by AutoSAT. During the search process, we employ multiple LLM agents to generate code, including:

Advisor. The **Advisor** reviews the solver code and provides a concise description of a specific part that requires improvement (e.g. its intuition, application, and comments for key variables), along with advice for further modifications to guide **Coder**.

Coder. The **Coder** receives the advice from the **Advisor** and the feedback from the **Evaluator**. It is responsible for modifying the actual heuristics within the SAT solver. Code is generated concurrently, exploring various optimization avenues by incorporating different modification directions suggested by the **Advisor** as well as insights from previous results.

Evaluator. Two issues may arise here: (1) writing incorrect code, and (2) writing identical code. We use a fault tolerance mechanism to detect erroneous code to ensure that the main program runs smoothly. After the code is generated, the **Evaluator** debugs it and categorizes each result as "Substantial Improvement", "Parameter Tuning", and "No Modification". Invalid code, such as those containing only variable name changes or errors, will be sent back to **Coder** for revision. Valid code, accompanied by a detailed description from the **Evaluator**, is then executed in the runtime environment, and the evaluation results are collected for the next iteration.

For these agents, we follow the instructions in OpenAI framework documentation (OpenAI 2023) and make appropriate adjustments and modifications based on actual usage. The prompt template obeys the following format:

- Define the **Role** of an agent as a solver expert who needs to assess and improve the heuristics in an SAT solver.
- Clearly state the **Goal**, such as providing optimization suggestions, writing code, or giving feedback.
- Enhance the agent's capabilities by inserting optional **Tips** that guide them to avoid common mistakes during code generation. Additionally, through this flexible interface, agents can effectively utilize external code and prior results, and can be instructed to specify the types of modifications, such as changing parameters, modifying heuristics, or adding new heuristics.
- Append the full SAT solver code at the end of each prompt to ensure that all agents share the same context.

Detailed prompt templates used in AutoSAT are available in appendix C.3.

4 Experimental Results

In this section, we present the experimental results of AutoSAT on 19 datasets, including improvements over the baseline solver, comparisons with SOTA solvers and their parameter-tuned variants, as well as the convergence behavior during the search process.

4.1 Settings

Environment SAT solvers are implemented in C++, and the interface to LLMs is implemented in Python. We compile the generated solver using g++ 9.4.0, and all experiments are conducted on three servers, each equipped with AMD EPYC 7763 64-core Processors. The budget $\mathcal{B}$, defined as the number of requests to the LLM for new heuristics, is set to 60 (larger than $0.6 * 9^2$, see Section 3.2) for testing AutoSAT on each dataset. The timeout bound $\mathcal{T}$ is set to 5000s for each SAT instance. Due to the superior performance of AutoSAT when using GPT-4o[2], we report only the results obtained with this model.

Benchmarks We select 19 datasets to explore the capabilities of AutoSAT, consisting of 14 datasets from SAT Competition 2023 (Balyo et al. 2023), and 5 datasets generated by Picat. For the SAT competition datasets, we filter out families with fewer than ten instances, and also exclude interval-matching and cryptography-simon because none of our baseline solvers could solve them within the 5000s timeout. This results in a total of 14 families of instances: argumentation, cryptography-ascon, set-covering-with-pairs (SCP), register-allocation, social-golfer, hashtable-safety, profitable-robust-product (PRP), brent-equations, mutilated-chessboard, satcoin, grs-fp-comm, school-timetabling, miter and tseitin. [3] The remaining 5 datasets are manually generated using the Picat tool, including CoinsGrid, LangFord, KnightTour, MineSweeper and Zamkeller, which can formulate constrained satisfaction problems into CNF formulas (Zhou et al. 2015). The sources and characteristics of all benchmark sets are summarized in Tab. 1. Further details on their generation are shown in Appendix B.5.

[2]We utilize the GPT-4o-2024-08-06 as default in this paper.
[3]In SCP, we collect 20 instances from SAT Competitions 2023, and 13 instances of the same group from SAT Competitions 2022 and 2018.

Table 1. **Dataset information**. We summarize datasets tested in this paper from SAT Competition 2023 (SC 2023 for short) and Picat (a language tool for problem generation). The size of each dataset and statistics on the number of variables and clauses are provided here, illustrating the diversity in problem complexity. More details are shown in Appendix B.4.

Dataset	Size	Source	variables mean	variables std	clauses mean	clauses std
cryptography-ascon	20	SC 2023	146,636	15,010	342,940	37,315
register-allocation	20	SC 2023	381	193	5,813	5,622
social-golfer	20	SC 2023	15,540	12,157	131,517	79,751
hashtable-safety	20	SC 2023	11,712,548	4,874,397	53,644,509	22,032,387
PRP	20	SC 2023	3,040,711	3,281,321	18,338,492	19,807,576
argumentation	20	SC 2023	962	190	27,960	23,034
SCP	33	SC 2023 et al.	682	164	26,969	10,830
brent-equations	19	SC 2023	64,488	22,997	274,022	98,596
mutilated-chessboard	12	SC 2023	35,124	15,784	1,418	335
satcoin	15	SC 2023	136,554	2,940	648,105	1,163
grs-fp-comm	17	SC 2023	433,277	363,536	1,252,138	1,047,668
school-timetabling	19	SC 2023	249,321	1,545	809,507	6,809
miter	11	SC 2023	33,732	73,283	110,363	237,151
tseitin	10	SC 2023	60,742	90,548	182,672	259,640
CoinsGrid	98	Picat	346,273	432,826	2,495,859	3,118,602
MineSweeper	88	Picat	618,801	531,657	9,065,224	7,909,645
KnightTour	56	Picat	223,697	313,704	10,692,883	17,482,123
LangFord	134	Picat	272,117	229,927	2,376,022	2,112,183
Zamkeller	80	Picat	24,592	22,190	310,804	335,102

To assess the generality of AutoSAT, we randomly split all benchmarks into training and testing datasets with a 7:3 ratio. The experiments based on different search strategies and parameter-tuning of baseline solvers are conducted exclusively on the training dataset. The best configuration is then evaluated on the test dataset to ensure an unbiased performance assessment.

Performance Metric In this paper, we consider two metrics for evaluating a SAT solver: (1) the Penalized Average Runtime with a factor of 2 score (PAR-2), and (2) the numbers of solved SAT and solved UNSAT instances within the given timeout bound. Detailed definitions of these metrics are provided in Appendix B.3. PAR-2 represents the average run time of a SAT solver for a set of instances, and for the instances where the solver cannot return a result within $\mathcal{T}$, the run time is penalized as $2\mathcal{T}$. A detailed explanation of the PAR-2 metric can be found in Appendix B.1. Both metrics have been widely used in SAT Competitions. We also use PAR-2 to evaluate the performance $f_P(A)$ of a solver A in Alg. 2 and Alg. 3.

4.2 Results

Competitive Results of AutoSAT Since the initial configuration of AutoSAT follows the default settings of EasySAT, we use EasySAT (Zhihan Chen 2022) as the baseline for the first comparison in this section. As shown in Tab. 2 and Tab. 3, LLMs can achieve significant improvement of EasySAT regarding PAR-2 across all datasets. Meanwhile, AutoSAT consistently outperforms the parameter tuning method SMAC3 (Lindauer et al. 2022) for

Table 2. **PAR-2 over training dataset.** This table reports the PAR-2 scores (lower is better) of different solvers across multiple benchmark datasets in the training phase. Compared solvers include EasySAT, EasySAT tune, AutoSAT_EA, and AutoSAT_GHC. For each dataset, PAR-2 is computed over all available instances, where unsolved cases are penalized by twice the cutoff time. The last column reports the relative PAR-2 reduction (%) of the better AutoSAT variant over EasySAT, computed as Improvement $= \frac{\text{EasySAT} - \min(\text{AutoSAT\_EA}, \text{AutoSAT\_GHC})}{\text{EasySAT}} \times 100\%$ (higher is better). The results highlight that AutoSAT variants consistently achieve lower PAR-2 on most datasets, demonstrating their stronger capability in handling both combinatorial and application-specific benchmarks.

Dataset	EasySAT	EasySAT tune	AutoSAT_EA	AutoSAT_GHC	improvement
argumentation	10000.00	10000.00	9300.21	**9000.21**	9.998%
cryptography-ascon	481.79	481.79	264.93	**224.57**	53.39%
register-allocation	10000.00	10000.00	**120.17**	10000.00	98.80%
social-golfer	10000.00	10000.00	7316.64	**7265.45**	27.35%
hashtable-safety	1191.14	1148.14	601.93	**585.43**	50.87%
PRP	5234.12	5136.43	**5042.46**	5112.54	3.66%
SCP	10000.00	10000.00	**6565.43**	9272.57	34.35%
brent-equations	10000.00	10000.00	**9387.85**	10000.00	6.12%
mutilated-chessboard	6385.25	6293.50	5449.75	**5432.37**	14.94%
satcoin	10000.00	10000.00	2036.80	**1757.60**	82.42%
grs-fp-comm	7834.91	7241.27	**7030.27**	7262.27	10.27%
school-timetabling	1856.08	989.23	**832.69**	849.08	55.13%
miter	5771.43	4636.57	**4299.43**	4305.27	25.48%
tseitin	8572.57	8572.57	**8571.71**	8572.00	0.01%
CoinsGrid	6886.97	6886.97	6120.99	**4785.29**	30.53%
MineSweeper	4116.84	3993.82	**1339.70**	1477.39	67.46%
KnightTour	8747.82	8259.28	**7981.41**	8494.97	8.76%
LangFord	8077.67	8049.81	**7337.16**	7364.55	9.16%
Zamkeller	3196.25	3177.38	2291.11	**1456.95**	54.42%

EasySAT. These observations clearly indicate that *LLMs is capable of designing better solvers than the provided baseline.*

Furthermore, we compare two other SAT solvers: the classical CDCL SAT solver MiniSat (Sörensson 2010) and the SOTA SAT solver Kissat (Fleury and Heisinger 2020). Both solvers are widely used in various SAT-related applications, and meanwhile are highly optimized for efficiency. We also apply SMAC3 (Lindauer et al. 2022) to these solvers to investigate how automated configuration further improves their performance (denoted as MiniSat tune and Kissat tune). Notably, the current version of Kissat[4] contains numerous heuristics and can select strategies based on predefined rules. More details about the two solvers are available in Appendix A. Our focus is primarily on MiniSat, which adds several strategies beyond EasySAT. Kissat, as a cutting-edge solver, goes far beyond EasySAT with many advanced techniques, setting a very high bar for EasySAT-based CDCL solvers.

Although EasySAT employs a basic CDCL framework in contrast to the complex and high-performance solvers MiniSat and Kissat, AutoSAT (obtained from EasySAT after modifications by LLMs) can achieve promising results

[4]We use version of Kissat 3.1.1 in https://github.com/arminbiere/kissat, and the version of MiniSat 2.2 in https://github.com/niklasso/minisat.

Table 3. **PAR-2 over testing dataset.** This table reports the PAR-2 scores (lower is better) of different solvers across multiple benchmark datasets in the testing phase. Compared solvers include EasySAT, EasySAT tune, AutoSAT_EA, and AutoSAT_GHC. For each dataset, PAR-2 is computed over all available instances, where unsolved cases are penalized by twice the cutoff time. The last column reports the relative PAR-2 reduction (%) of the better AutoSAT variant over EasySAT, computed as Improvement $= \frac{\text{EasySAT} - \min(\text{AutoSAT\_EA}, \text{AutoSAT\_GHC})}{\text{EasySAT}} \times 100\%$ (higher is better). The results demonstrate that AutoSAT variants achieve competitive or superior performance on several datasets, confirming the generalization ability of our approach beyond the training benchmarks.

Dataset	EasySAT	EasySAT tune	AutoSAT_EA	AutoSAT_GHC	improvement
argumentation	10000.00	10000.00	**8577.83**	10000.00	14.22%
cryptography-ascon	226.33	226.33	208.83	**169.50**	25.11%
register-allocation	10000.00	10000.00	**1819.00**	10000.00	81.81%
social-golfer	10000.00	10000.00	5849.67	**5027.83**	49.72%
hashtable-safety	811.17	798.83	**728.10**	767.67	10.23%
PRP	5156.67	5101.17	**5021.33**	5102.71	2.63%
SCP	10000.00	10000.00	**5387.00**	8762.20	46.13%
brent-equations	10000.00	10000.00	10000.00	10000.00	0.00%
mutilated-chessboard	10000.00	10000.00	10000.00	**8525.36**	14.75%
satcoin	10000.00	10000.00	2025.60	**1623.16**	83.77%
grs-fp-comm	5670.50	5605.50	**5328.67**	5443.71	6.03%
school-timetabling	1853.83	1811.00	**1620.33**	1737.33	12.60%
miter	5019.75	5017.75	**5015.75**	5016.75	0.08%
tseitin	10000.00	10000.00	10000.00	10000.00	0.00%
CoinsGrid	6432.40	6432.40	6140.10	**5972.47**	7.15%
MineSweeper	2546.33	2201.48	**1277.22**	1349.59	49.87%
KnightTour	9411.76	8881.24	**8842.06**	8925.82	6.05%
LangFord	8460.37	8382.59	7856.63	**7797.39**	7.83%
Zamkeller	4744.00	4494.67	4068.21	**3219.08**	32.14%

that are comparable to both solvers. More precisely, AutoSAT outperforms MiniSat and its parameter-tuned variants on 12 of the 19 datasets, and it outperforms Kissat and its parameter-tuned variants on 4 datasets.

Fine-grained search space enables LLMs to construct better solvers. Unlike EoH which utilizes LLMs to generate optimization solvers from scratch (F. Liu et al. 2024), AutoSAT searches for improved solvers by working on a fine-grained 9 dimensional space. This approach allows AutoSAT to optimize SAT solvers with long text code more efficiently. We also attempted to follow the approach of EoH (F. Liu et al. 2024) to generate a SAT solver from scratch, but this only produced a basic CDCL framework which could not compete with MiniSat and Kissat. In contrast, experimental results show that AutoSAT is able to achieve comparable performance with MiniSat and Kissat.

Furthermore, the fine-grained search space can serve as a bridge that connects existing benchmarking and theoretical research on discrete optimization to LLM-based optimization methods. For example, we draw on the theoretical analysis of LeadingOnes (C. Doerr, F. Ye, Rijn, et al. 2018) to set up our budget, as detailed in Section 3.2.

Table 4. **PAR-2 over the training dataset.** This table reports the PAR-2 scores (lower is better) of EasySAT, MiniSat, MiniSat tune, Kissat, Kissat tune, and AutoSAT across multiple benchmark datasets in the training phase. For AutoSAT, we report the better result between AutoSAT_EA and AutoSAT_GHC for each dataset.

Dataset	EasySAT	MiniSat	MiniSat tune	Kissat	Kissat tune	AutoSAT
argumentation	10000.00	8802.36	8371.14	3894.86	3677.57	9000.21
cryptography-ascon	481.79	184.14	234.21	157.79	158.86	224.57
register-allocation	10000.00	10000.00	10000.00	8407.93	5503.57	120.17
social-golfer	10000.00	9299.86	9286.57	8600.50	6872.50	7265.45
hashtable-safety	1191.14	211.93	217.29	518.57	287.21	585.43
PRP	5234.12	5079.71	5075.86	5021.43	5014.07	5042.46
SCP	10000.00	6226.61	6530.48	86.48	38.09	6565.43
brent-equations	10000.00	9387.85	9387.85	837.70	770.15	9387.85
mutilated-chessboard	6385.25	10000.00	10000.00	3244.25	3242.50	5432.37
satcoin	10000.00	10000.00	10000.00	1336.60	884.60	1757.60
grs-fp-comm	7834.91	450.34	249.08	3157.97	3144.00	7030.27
school-timetabling	1856.08	4865.82	6923.08	1565.88	788.80	832.69
miter	5771.43	4309.06	4302.29	1435.00	1434.11	4299.43
tseitin	8572.57	8571.71	8571.71	8571.71	8571.71	8571.71
CoinsGrid	6886.97	5527.88	5573.68	4033.06	3331.18	4785.29
MineSweeper	4116.84	78.82	7.75	120.44	45.62	1339.70
KnightTour	8747.82	9010.74	8462.59	8466.33	8468.97	7981.41
LangFord	8077.67	8543.82	8584.00	5164.95	4832.54	7337.16
Zamkeller	3196.25	5214.88	4697.30	1589.36	392.18	1456.95

Useful Searching History Using the register-allocation dataset as an example, we illustrate the search process of AutoSAT with (1 + 1) EA in Fig. 1. Benefiting from the fine-grained search, we can easily observe that, among the $\mathcal{B} = 60$ candidates generated by the LLMs, AutoSAT achieved improvements in 10 evaluations, including:

- updating *reduce* heuristic function 2 times;
- updating *restart_condition* heuristic function 3 times;
- updating *rephase_condition* heuristic function 3 times;
- updating *rephase* heuristic function 2 times;
- updating *heuristics_bump_var* heuristic function 1 time;
- updating *reduce_condition* heuristic function 2 times.

Note that AutoSAT with EA as a search strategy supports modifying multiple heuristics simultaneously. Precise update information for each heuristic function can be valuable for future adaptive searching strategies, as it reflects which heuristics have a significant impact on the solver performance for specific instances. In addition, compared to approaches that generate code from scratch, this final solver obtained through this process provides clearer guidance and more useful knowledge for SAT researchers.

Table 5. **PAR-2 over testing dataset.** This table reports the PAR-2 scores (lower is better) of EasySAT, MiniSat, MiniSat tune, Kissat, Kissat tune, and AutoSAT across multiple benchmark datasets in the testing phase. For AutoSAT, we report the better one from AutoSAT_EA and AutoSAT_GHC for each dataset.

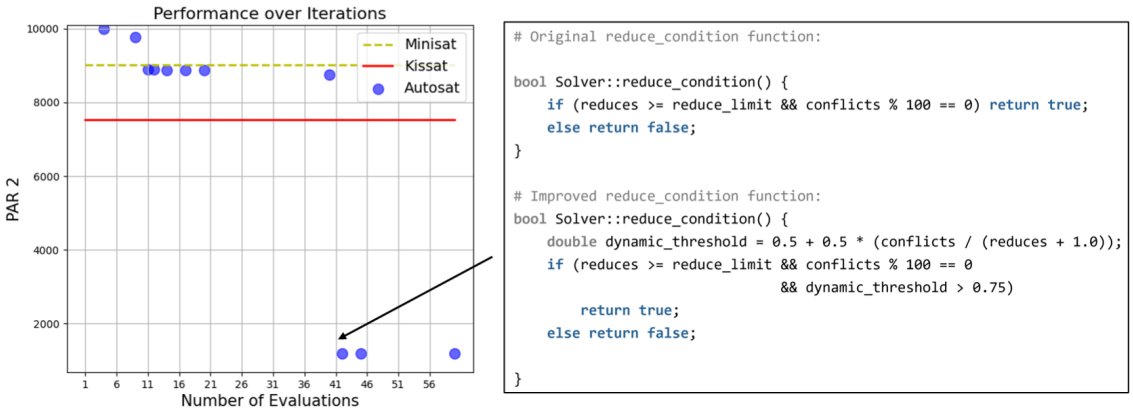
Dataset	EasySAT	MiniSat	MiniSat tune	Kissat	Kissat tune	AutoSAT
argumentation	10000.00	8605.33	10000.00	2008.33	**1751.50**	8577.83
Cryptography-ascon	226.33	189.83	188.17	**123.17**	141.67	169.50
register-allocation	10000.00	10000.00	10000.00	8890.00	8353.50	**1819.00**
social-golfer	10000.00	5578.17	8553.00	5053.33	5585.17	**5027.83**
hashtable-safety	811.17	161.17	**159.50**	567.17	310.50	728.10
PRP	5156.67	5049.00	5079.33	5012.33	**5012.00**	5021.33
SCP	10000.00	6142.65	6312.50	52.10	**18.60**	5387.00
brent-equations	10000.00	10000.00	10000.00	**74.17**	1667.50	10000.00
mutilated-chessboard	10000.00	8588.64	10000.00	3244.25	**3242.50**	8525.36
satcoin	10000.00	10000.00	10000.00	1336.60	**884.60**	1623.16
grs-fp-comm	5670.50	5225.17	5124.54	2160.17	**2144.17**	5328.67
school-timetabling	1853.83	3047.46	5800.17	1693.67	1684.33	**1620.33**
miter	5019.75	5020.43	5014.50	5003.75	**5003.23**	5015.75
tseitin	10000.00	10000.00	10000.00	10000.00	10000.00	10000.00
CoinsGrid	6432.40	6010.13	6012.47	3980.70	**2879.83**	5972.47
MineSweeper	2546.33	32.07	**6.93**	66.85	38.52	1277.22
KnightTour	9411.76	9411.76	8835.94	9411.76	9025.35	**8842.06**
LangFord	8460.37	9070.85	8950.20	6256.44	**6143.51**	7797.39
Zamkeller	4744.00	6888.67	6344.79	3256.00	**200.29**	3219.08

4.3 Analysis of Discovered Heuristics

In this section, to demonstrate LLMs' ability of generating effective heuristics, we provide more contrastive examples (one for each function candidate), along with the explanations of the changes. Specific code implementations for each heuristic are provided in Appendix D.

Modifying *reduce* Function (see Figs. 2 and 3 for the original and discovered heuristics)**.** The discovered heuristics introduce a multi-tiered adaptive reduction strategy that dynamically adjusts clause removal probabilities according to LBD value, age, and activity. Unlike the original's fixed 50% removal for $lbd \geq 5$ clauses, the new approach uses: (1) LBD-tiered thresholds (80% removal for $lbd \geq 8$, 50% for $lbd \geq 5$, 20% for $lbd \geq 3$), (2) age-weighted scaling via $1 - e^{-(old\_size-i)/1000}$ to prioritize recent clauses, and (3) adaptive reduce_limit growth to balance reduction frequency. These modifications enhance clause database quality by pruning ineffective clauses while preserving structurally important ones, thereby reducing memory usage and propagation costs without compromising proof completeness.

Modifying *rephase* function (see Fig. 4 for the original and discovered heuristics)**.** The discovered heuristics replace static parameters with dynamic adaptive strategies for phase selection and threshold adjustment. Instead of merely decaying the threshold linearly (threshold $*= 0.9$) and increasing rephase_limit by a fixed amount (rephase_limit +8192), it scales rephase_limit multiplicatively and additively (rephase_limit $*1.3 + 2048$) to better




```
# Original reduce_condition function:

bool Solver::reduce_condition() {
    if (reduces >= reduce_limit && conflicts % 100 == 0) return true;
    else return false;
}

# Improved reduce_condition function:
bool Solver::reduce_condition() {
    double dynamic_threshold = 0.5 + 0.5 * (conflicts / (reduces + 1.0));
    if (reduces >= reduce_limit && conflicts % 100 == 0
                                && dynamic_threshold > 0.75)
        return true;
    else return false;

}
```

Fig. 1. **PAR-2 during searching process in register-allocation dataset**. This figure illustrates the search trajectory of AutoSAT towards better heuristics. Blue dots indicate the PAR-2 values obtained by the explored solver candidates, which show improvements along the optimization process. We highlight on the right a modification of the *reduce_condition_function* heuristics, which happens at the ninth evaluation.

match solver progress. The discovered heuristics also introduce a progress-dependent phase-saving strategy that hybridizes activity-based heuristics, local-best exploitation with probabilistic exploration (15% randomization), and strong diversification (inverting local best) based on conflict-driven progress (conflicts / (conflicts + 10000)). This adaptivity balances exploitation of learned information and exploration of new phases, potentially improving search efficiency and reducing stagnation. Additionally, the threshold is dynamically set relative to trail size and search progress (trail.size() ∗(0.7 + 0.3∗progress)), enabling more context-aware restart behavior compared to the fixed decay in the original heuristics.

Modifying *heuristics bump var* function (see Fig. 5 for the original and discovered heuristics)**.** The discovered heuristics introduce a decay mechanism and a level-based bonus system, which collectively improve the solver's ability to balance exploration and exploitation during search. Specifically, it replaces the static bump value with a dynamic weight incorporating: (1) a level bonus (1.2×) for variables at or above backtrackLevel, prioritizing those directly involved in recent conflicts, and (2) an exponential decay factor that reduces influence for frequently bumped variables, preventing overemphasis and promoting diversification. This approach refines variable activity management by adaptively scaling bump magnitudes based on both structural relevance (level) and historical activity (decay), leading to more informed branching decisions and potentially faster convergence through improved conflict analysis and clause learning.

Modifying *restart condition* function (see Fig. 6 for the original and discovered heuristics). The discovered heuristics introduce a dynamic threshold mechanism that incorporates both the conflict rate and the variance of LBD values, replacing the original static threshold of 0.8 with a more sensitive base factor of 0.65 and a dynamically adjusted component. This dynamic threshold, calculated as $1.0 + \frac{\text{conflict rate} \times \text{lbd variance}}{100}$, allows the restart condition to adapt to the solver's current state: it increases the effective threshold during high-variance LBD phases (indicating unstable search) and high conflict rates, promoting restarts when the search is less productive while reducing unnecessary restarts during stable low-variance phases. This results in a more nuanced

and efficient restart strategy, potentially improving the solver's ability to escape local minima and reduce overall runtime by aligning restarts more closely with actual search difficulty.

Modifying *restart condition update* function (see Fig. 7 for the original and discovered heuristics). The discovered heuristics replace the static cumulative sum for slow_lbd_sum with an exponentially weighted moving average using weights FAST_WEIGHT = 0.8 (recent bias) and SLOW_WEIGHT = 0.2 (historical bias), which prioritizes recent LBD values while retaining memory of past behavior, leading to more adaptive restart decisions. Additionally, it introduces dynamic scaling factors (1.05× for lbd $<$ 10, 0.95× for lbd $>$ 30) that adjust slow_lbd_sum based on LBD severity, amplifying the effect of low-LBD clauses (likely high-quality) and dampening high-LBD clauses (likely redundant). This combined approach improves the response to solver state changes, reduces overfitting to historical data, and optimizes the exploration-exploitation balance during restarts, potentially boosting solver performance on structured problems.

Modifying *restart* function (see Fig. 8 for the original and discovered heuristics). The discovered heuristics introduce a dynamic restart strategy that calculates fast (recent) and slow (historical) LBD averages to adjust restart aggressiveness on-the-fly, triggering more aggressive restarts (backtracking 70% of the trail) when the LBD becomes worse and less aggressive ones (backtracking 90%) when it improves, unlike the original's fixed full restart to level 0. This adapts to the problem's structure by preserving useful learned clauses during partial restarts, reducing unnecessary backtracking. Furthermore, it refines phase selection with a VSIDS-based heuristic (15% probability) that leverages variable activity scores, along with the optimized probabilities for local best (40%), inverted local best (30%), and random phases (15%), replacing the original's rigid split. These changes improve convergence by aligning restarts with search progress and leveraging conflict-driven data for phase decisions, enhancing both diversification and intensification.

Modifying *rephase condition update* function (see Fig. 9 for the original and discovered heuristics.) The discovered heuristics introduce a dynamic threshold and a solver activity metric, moving beyond the original's static limit check. The original condition, rephases $\geq$ rephase_limit, triggers rephasing based solely on a fixed count, whereas the improved version employs a dynamic threshold scaled by 1.5 (rephase_limit * 1.5) and integrates a conflict-to-restart ratio (conflict_restart_ratio > 1.8). This dual requirement ensures that rephasing only occurs during periods of high solver activity, as indicated by an elevated number of conflicts relative to restarts, which prevents unnecessary and computationally expensive phase resetting during stable search periods. Consequently, this adaptation optimizes the timing of rephasing, reducing overhead and potentially improving the solver's overall performance by aligning the restart strategy more closely with its current search state.

Modifying *reduce condition update* function (see Fig. 10 for the original and discovered heuristics.) The discovered heuristics incorporate a dynamic, multi-faceted approach that considers both memory pressure (quantified as memory_pressure = $\frac{\text{clause\_DB.size()}}{\text{origin\_clauses+10000}}$) and search progress (measured by progress_rate = $\frac{\text{conflicts}}{\text{reduce\_limit+1000}}$). This allows the solver to adaptively trigger clause database reduction under high memory pressure ($>$ 2.0) or slow progress ($>$ 0.8) by lowering the reduction threshold to 70% of reduce_limit, while in normal operation it requires both the original threshold and moderate memory pressure ($>$ 1.2) to activate. This optimization improves memory management, prevents premature stagnation, and maintains search efficiency by dynamically balancing learning effort and resource constraints based on real-time solver state.

Modifying *bump var* function (see Fig. 11 for the original and discovered heuristics). The discovered heuristics incorporate a decay factor (decay_factor = 0.95) into the activity update rule, transforming it from a purely additive model (activity[var] $\leftarrow$ activity[var] + var_inc $\times$ coeff) to an exponential moving average (activity[var] $\leftarrow$ activity[var] $\times$ decay_factor + var_inc $\times$ coeff). This modification ensures that older activity contributions gradually diminish over time, prioritizing recent variable engagements in conflicts and better aligning with the VSIDS heuristic's objective of focusing on variables involved in recent conflicts. The decay

mechanism mitigates unbounded activity growth, reduces the frequency of overflow-induced rescaling operations (which disrupt score consistency and require $O(n)$ time) and enhances the solver's adaptability to dynamic search patterns by continuously refining variable priorities.

In summary, by integrating concepts like exponential moving averages, probabilistic phase exploration, and memory-pressure-aware reduction, the discovered heuristics establish feedback mechanisms that balance diversification with intensification while avoiding unnecessary computations. This offers a template for future SAT research, wherein LLMs act as heuristic generators and human experts concentrate on analysis and integration.

5 Conclusion and Discussion

We introduce AutoSAT, a framework that leverages LLMs to optimize the heuristic functions of CDCL solvers. In particular, AutoSAT is built on a lightweight CDCL solver EasySAT. Unlike traditional LLM-based optimization approaches that focus on generating code from scratch, AutoSAT integrates expert knowledge to guide LLMs in refining specific heuristic functions, thereby enabling the automatic production of competitive SAT solvers. Specifically, we employ GHC and $(1 + 1)$ EA as search strategies to iteratively select and optimize these heuristic functions via LLMs. Empirically, AutoSAT outperforms MiniSat and its parameter-tuned variants on 12 of the 19 datasets, and outperforms Kissat and its parameter-tuned variants on 4 datasets.

This work presents a novel paradigm for LLM-driven SAT solver optimization which focuses on a possibly infinite-dimensional, fine-grained search space within a CDCL solver. This paradigm facilitates a deeper understanding of the LLM-based optimization process. A practical limitation of AutoSAT is the computational cost of the search process, which is dominated by solver evaluations on the training dataset. However, this cost is incurred only offline and largely amortized once a niche solver has been found for a target dataset. Moreover, the search cost may be reduced in several ways: by using smaller timeouts during search, evaluating on representative subsets of highly structured instance families, or using low-cost signals from intermediate solver behavior (e.g., decision depth) to filter weak candidates before full evaluation.

Beyond its capability to produce competitive SAT solvers, the AutoSAT framework also opens up several promising directions for future research. Firstly, more advanced modules can be incorporated into the CDCL solvers to enhance their performance. Secondly, self-adaptive search strategies offer further opportunities to improve solver efficiency. Lastly, a deeper investigation into the behavior of LLMs will also be crucial for understanding and improving the performance of LLM-based solvers. Therefore, we intend to investigate how and why LLMs modify heuristics, with the goal of establishing practical guidelines for LLM-based optimization.

References

J. Achiam et al.. 2023. "GPT-4 Technical Report." *arXiv preprint*, arXiv:2303.08774.

S. Adriaensen, A. Biedenkapp, G. Shala, N. Awad, T. Eimer, M. Lindauer, and F. Hutter. 2022. "Automated dynamic algorithm configuration." *Journal of Artificial Intelligence Research*, 75, 1633–1699.

G. Audemard, J.-M. Lagniez, B. Mazure, and L. Sais. 2010. "Boosting local search thanks to CDCL." In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 474–488.

G. Audemard and L. Simon. 2012. "Glucose 2.1: Aggressive, but reactive, clause database management, dynamic restarts (system description)." In: *Pragmatics of SAT 2012*.

G. Audemard and L. Simon. 2009. "Predicting learnT clauses quality in modern SAT solvers." In: *the 21st International Joint Conference on Artificial Intelligence*. Vol. 9, 399–404.

T. H. Bäck et al.. 2023. "Evolutionary algorithms for parameter optimization—thirty years later." *Evolutionary Computation*, 31, 2, 81–122.

A. Balint, A. Belov, M. Järvisalo, and C. Sinz. 2015. "Overview and analysis of the SAT Challenge 2012 solver competition." *Artificial Intelligence*, 223, 120–155.

A. Balint and A. Fröhlich. 2010. "Improving stochastic local search for SAT with a new probability distribution." In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 10–15.

T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, (Eds.) . 2023. *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. English. Department of Computer Science, University of Helsinki.

A. Biere. 2019. "CaDiCaL at the SAT Race 2019." In: *Proceedings of SAT Competition 2019: Solver and Benchmark Descriptions*. Vol. 2019. Department of Computer Science, University of Helsinki, 8–9.

A. Biere. 2010. "Lingeling, plingeling, picosat and precosat at sat race 2010." *FMVReport Series Technical Report*, 10.

A. Biere and A. Fröhlich. 2015. "Evaluating CDCL restart schemes." In: *Proceedings of Pragmatics of SAT*, 1–17.

A. Biere, M. Heule, H. van Maaren, and T. Walsh. 2009. "Conflict-driven clause learning SAT solvers." *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, 4, 131–153.

S. Böttcher, B. Doerr, and F. Neumann. 2010. "Optimal fixed and adaptive mutation rates for the LeadingOnes problem." In: *International Conference on Parallel Problem Solving from Nature*. Springer, 1–10.

T. Brown et al.. 2020. "Language models are few-shot learners." *Advances in neural information processing systems*, 33, 1877–1901.

S. Cai, C. Luo, and K. Su. 2015. "CCAnr: A Configuration Checking Based Local Search Solver for Non-random Satisfiability." In: *Theory and Applications of Satisfiability Testing*. Springer, 1–8.

S. Cai and X. Zhang. 2021. "Deep cooperation of CDCL and local search for SAT." In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 64–81.

W. Chen et al.. 2023. "Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors." In: *the 12th International Conference on Learning Representations*.

M. S. Cherif, D. Habet, and C. Terrioux. 2021. "Kissat mab: Combining vsids and chb through multi-armed bandit." *SAT Competition*, 2021, 15.

S. A. Cook. 1971. "The complexity of theorem-proving procedures." In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, 151–158.

B. Doerr and F. Neumann. 2019. *Theory of evolutionary computation: Recent developments in discrete optimization*. Springer Nature.

C. Doerr, F. Ye, N. Horesh, H. Wang, O. Shir, and T. Back. 2020. "Benchmarking discrete optimization heuristics with IOHprofiler." *Applied Soft Computing*, 88, 106027.

C. Doerr, F. Ye, S. van Rijn, H. Wang, and T. Bäck. 2018. "Towards a theory-guided benchmarking suite for discrete black-box optimization heuristics: profiling (1+λ) EA variants on OneMax and LeadingOnes." In: *Proceedings of the Genetic and Evolutionary Computation Conference*, 951–958.

S. Droste, T. Jansen, and I. Wegener. 2002. "On the analysis of the (1+1) evolutionary algorithm." *Theoretical Computer Science*, 276, 1-2, 51–81.

A. Fleury and M. Heisinger. 2020. "Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020." In: *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*. Vol. 2020. Department of Computer Science, University of Helsinki, 51–53.

P. Guo, F. Liu, X. Lin, Q. Zhao, and Q. Zhang. 2024. *L-autoda: Leveraging large language models for automated decision-based adversarial attacks*. (2024). eprint: arXiv:2401.15335.

N. Hansen and R. Ros. 2010. "Black-box optimization benchmarking of NEWUOA compared to BIPOP-CMA-ES: on the BBOB noiseless testbed." In: *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*, 1519–1526.

H. H. Hoos, F. Hutter, and K. Leyton-Brown. 2021. "Automated configuration and selection of SAT solvers." In: *Handbook of Satisfiability*. IOS Press, 481–507.

H. H. Hoos and T. Stützle. 2000. "Local search algorithms for SAT: An empirical evaluation." *Journal of Automated Reasoning*, 24, 4, 421–481.

R. G. Jeroslow and J. Wang. 1990. "Solving propositional satisfiability problems." *Annals of mathematics and Artificial Intelligence*, 1, 1, 167–187.

M. Josifoski et al.. 2023. *Flows: Building blocks of reasoning and collaborating ai*. (2023). eprint: arXiv:2308.01285.

P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann. 2019. "Automated algorithm selection: Survey and perspectives." *Evolutionary computation*, 27, 1, 3–45.

S. Kochemazov, O. Zaikin, V. Kondratiev, and A. Semenov. 2019. "MapleLCMDistChronoBT-DL, duplicate learnts heuristic-aided solvers at the SAT Race 2019." In: *Proceedings of SAT Competition 2019: Solver and Benchmark Descriptions*. Vol. 2019. Department of Computer Science, University of Helsinki, Finland, 24–24.

O. Kullmann. 2021. "Fundaments of branching heuristics." In: *Handbook of Satisfiability*. IOS Press, 351–390.

R. Li et al.. 2023. "Starcoder: may the source be with you!" *arXiv preprint*, arXiv:2305.06161.

J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. 2016. "Learning rate based branching heuristic for SAT solvers." In: *Theory and Applications of Satisfiability Testing–SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings 19*. Springer, 123–140.

J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart. 2016. "Maple-comsps, maplecomsps lrb, maplecomsps chb." *Proceedings of SAT Competition*, 2016, 66–72.

J. H. Liang, C. Oh, M. Mathew, C. Thomas, C. Li, and V. Ganesh. 2018. "Machine learning-based restart policy for CDCL SAT solvers." In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 94–110.

M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, D. Deng, C. Benjamins, T. Ruhkopf, R. Sass, and F. Hutter. 2022. "SMAC3: A versatile Bayesian optimization package for hyperparameter optimization." *Journal of Machine Learning Research*, 23, 54, 1–9.

F. Liu, T. Xialiang, M. Yuan, X. Lin, F. Luo, Z. Wang, Z. Lu, and Q. Zhang. 2024. "Evolution of Heuristics: Towards Efficient Automatic Algorithm Design Using Large Language Model." In: *the 41st International Conference on Machine Learning*.

H. Liu, P. Xu, Y. Pu, L. Yin, H.-L. Zhen, M. Yuan, T.-Y. Ho, and B. Yu. 2024. "Neuroselect: Learning to select clauses in sat solvers." In: *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 1–6.
M. Luby, A. Sinclair, and D. Zuckerman. 1993. "Optimal speedup of Las Vegas algorithms." *Information Processing Letters*, 47, 4, 173–180.
C. Luo, H. Hoos, and S. Cai. 2020. "PbO-CCSAT: Boosting local search for satisfiability using programming by optimisation." In: *International Conference on Parallel Problem Solving from Nature*. Springer, 373–389.
Y. S. Mahajan, Z. Fu, and S. Malik. 2004. "Zchaff2004: An efficient sat solver." In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 360–375.
M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. "Chaff: Engineering an efficient SAT solver." In: *Proceedings of the 38th annual Design Automation Conference*, 530–535.
D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers. 2024. "Using an LLM to help with code understanding." In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.
H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian. 2023. "A comprehensive overview of large language models." *arXiv preprint*, arXiv:2307.06435.
OpenAI. 2023. *OpenAI API Documentation*. https://platform.openai.com/docs. (2023).
A. Ramos, P. Van Der Tak, and M. J. Heule. 2011. "Between restarts and backjumps." In: *Theory and Applications of Satisfiability Testing-SAT 2011: 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings 14*. Springer, 216–229.
B. Romera-Paredes et al.. 2024. "Mathematical discoveries from program search with large language models." *Nature*, 625, 7995, 468–475.
D. Selsam and N. Bjørner. 2019. "Guiding high-performance SAT solvers with unsat-core predictions." In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 336–353.
D. Selsam, M. Lamm, B. Bünz, P. Liang, D. L. Dill, and L. De Moura. 2019. "Learning a SAT solver from single-bit supervision." In: *7th International Conference on Learning Representations, ICLR 2019*.
J. M. Silva and K. A. Sakallah. 1996. "GRASP-a new search algorithm for satisfiability." In: *Proceedings of International Conference on Computer Aided Design*. IEEE, 220–227.
"Strangenight." *Proceedings of SAT Competition 2013: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science, University of Helsinki, 89–90.
N. Sörensson. 2010. "Minisat 2.2 and minisat++ 1.1." *A short description in SAT Race*, 2010.
M. de Souza and M. Ritt. 2018. "Automatic grammar-based design of heuristic algorithms for unconstrained binary quadratic programming." In: *European Conference on Evolutionary Computation in Combinatorial Optimization*. Springer, 67–84.
D. Speck, A. Biedenkapp, F. Hutter, R. Mattmüller, and M. Lindauer. 2021. "Learning heuristic selection with dynamic algorithm configuration." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 31, 597–605.
N. van Stein and T. Bäck. 2024. "LLaMEA: A Large Language Model Evolutionary Algorithm for Automatically Generating Metaheuristics." *arXiv preprint*, arXiv:2405.20132.
P. Vaezipoor, G. Lederman, Y. Wu, R. Grosse, and F. Bacchus. 2020. "Learning clause deletion heuristics with reinforcement learning." In: *5th Conference on Artificial Intelligence and Theorem Proving*, 80.
W. Wang, Y. Hu, M. Tiwari, S. Khurshid, K. McMillan, and R. Miikkulainen. 2021. "NeuroBack: Improving CDCL SAT Solving using Graph Neural Networks." *arXiv preprint*, arXiv:2110.14053.
Q. Wu et al.. 2023. "AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework." *arXiv preprint*, arXiv:2308.08155.
H. Yang, S. Yue, and Y. He. 2023. "Auto-GPT for online decision making: Benchmarks and additional opinions." *arXiv preprint*, arXiv:2306.02224.
Y. Yao, F. Liu, J. Cheng, and Q. Zhang. 2024. "Evolve cost-aware acquisition functions using large language models." In: *International Conference on Parallel Problem Solving from Nature*. Springer, 374–390.
F. Ye, H. Wang, C. Doerr, and T. Bäck. 2020. "Benchmarking a genetic algorithm with configurable crossover probability." In: *International Conference on Parallel Problem Solving from Nature*. Springer, 699–713.
H. Ye, J. Wang, Z. Cao, and G. Song. 2024. "ReEvo: Large Language Models as Hyper-Heuristics with Reflective Evolution." *arXiv preprint*, arXiv:2402.01145.
R. Zhang, F. Liu, X. Lin, Z. Wang, Z. Lu, and Q. Zhang. 2024. "Understanding the Importance of Evolutionary Search in Automated Heuristic Design with Large Language Models." In: *International Conference on Parallel Problem Solving from Nature*. Springer, 185–202.
X. Z. Zhihan Chen. 2022. *EasySAT: A Simple CDCL SAT Solver*. https://github.com/shaowei-cai-group/EasySAT. (2022).
N.-F. Zhou, H. Kjellerstrand, and J. Fruhman. 2015. *Constraint solving and planning with Picat*. Vol. 11. Springer.

A Solver Description

AutoSAT optimizes the heuristic functions of CDCL solvers over a fine-grained search space. We design and implement a CDCL solver comprising nine independent heuristic functions, with their initial settings adopted from the classic EasySAT solver[5]. EasySAT is a CDCL solver integrated with the following techniques beyond the basic framework provided in Algorithm 1:

- **Branching Heuristics**: EasySAT implements the Variable State Independent Decaying Sum (VSIDS) heuristic, which prioritizes variables based on their recent conflict involvement and focus on the most relevant parts of the search space.
- **Restart and Rephase**: Restart strategies implemented in EasySAT help escape difficult regions of the search space, while rephase strategies periodically flip variable phases to explore different search paths and accelerate convergence.
- **Clause Reduction**: EasySAT periodically removes clauses deemed unlikely to contribute to a solution helps manage memory and focus on useful clauses.

Compared with EasySAT, MiniSat introduces several additional techniques, including:

- **Preprocessing and Simplification**: MiniSat applies various preprocessing techniques to simplify the formula before running the main solving process.
- **Data Structure Optimizations**: MiniSat optimizes data structure so that it can handle large-scale instances and process millions of clauses and variables efficiently.

Compared with EasySAT and MiniSat, Kissat implements a range of advanced techniques that significantly improve its performance, establishing it as a SOTA solver in SAT competitions. Several key improvements are listed below:

- **Clause Management and Database Reduction**: Kissat introduces more sophisticated methods for managing learned clauses, including glue clause recycling, which measures the quality of learned clauses and removes less useful ones. This approach ensures that the solver maintains a smaller, more relevant set of clauses.
- **Enhanced Restart Strategies**: Kissat uses tiered restart strategies, where the solver decides when to restart based on multiple levels of heuristics. This approach improves the balance between local search and global search, leading to faster convergence on difficult problems.
- **Data Structure Optimization**: Kissat further optimizes memory usage and data access with highly efficient bit-level compression and cache-friendly designs, which reduce the overhead associated with clause storage and propagation.
- **In-Search Simplification**: Kissat continuously applies in-search simplification techniques, including variable and clause elimination, to dynamically reduce the complexity of the problem as the search progresses. This ensures that the solver remains efficient throughout the entire solving process.
- **Heuristic Enhancements**: Kissat utilizes hybrid and dynamic heuristics for variable selection, such as combining VSIDS with Conflict History-Based (CHB) heuristics, which adapt to different phases of the solving process to enhance decision-making.

B Experiment Details

B.1 PAR-2

The PAR-2 (Penalized Average Runtime with a factor of 2) score is used to evaluate the performance of SAT solvers over a benchmark set of instances. It is particularly useful when solvers fail to return a result within a given timeout bound. The PAR-2 score penalizes such instances with twice the timeout bound $\mathcal{T}$.

[5]https://github.com/shaowei-cai-group/EasySAT

Consider a benchmark set of n instances. Let t_i be the runtime of the SAT solver on instance i. The PAR-2 score is defined as:

$$\text{PAR-2} = \frac{1}{n}\sum_{i=1}^{n}\tau_i,$$

where

$$\tau_i = \begin{cases} t_i, & \text{if } t_i \le \mathcal{T}, \\ 2\mathcal{T}, & \text{if } t_i > \mathcal{T} \text{ or the solver fails to return a result.} \end{cases}$$

For an illustration, consider an example with 3 instances and a timeout bound $\mathcal{T} = 100$ seconds. The runtimes (in seconds) for each instance are: $t_1 = 80$ for the first instance, $t_2 = 120$ for the second instance, and the solver fails to return a result within $\mathcal{T}$ for the third instance. Therefore,

- $\tau_1 = 80$ since $t_1 \le \mathcal{T}$,
- $\tau_2 = 200$ (penalized) since $t_2 > \mathcal{T}$,
- $\tau_3 = 200$ (penalized) since the solver fails.

Then the PAR-2 score is calculated as:

$$\text{PAR-2} = \frac{1}{3}(80 + 200 + 200) = \frac{480}{3} = 160.$$

B.2 Search Space for Parameter Tuning

In this paper, we also use SMAC3 to optimize the parameters of the baseline SAT solvers for different datasets. The search space for each solver, including the parameter name, variable type, description, and variable domains, is presented in Tables 6, 7, 8.

Table 6. EasySAT configuration parameters

Parameter	Type	Description	Search Space
`var-decay`	double	Variable activity decay factor	{0.75, 0.8, 0.85}
`rephase-inc`	int	Rephasing counter increment threshold	{50, 55, 60, 65, 70}
`lbd-size`	int	LBD (Literal Block Distance) threshold for clause deletion	{50, 60, 70, 80}
`restart-factor`	double	Dynamic restart scaling factor	{0.7, 0.75, 0.8, 0.85}
`bad-clause-size`	int	Size threshold for identifying bad clauses	{3, 4, 5, 6}
`reduce-interval`	int	Clause database reduction interval	{4096, 7168, 8192, 9216}
`rephase-interval`	int	Rephasing execution interval	{4096, 7168, 8192, 9216}

B.3 More Details about Experiments

Tables 9–12 provide the instance sizes of the training and testing sets for each dataset, along with the numbers of solved SAT and UNSAT instances corresponding to the results in Tables 2–5. These additional details offer a more comprehensive understanding of the experimental data and outcomes.

B.4 Dataset Description

We provide a brief overview of the benchmark datasets, which include 14 datasets from the SAT Competition 2023 and 5 newly generated datasets.

Table 7. MiniSat configuration parameters

Parameter	Type	Description	Search Space
`var-decay`	double	Variable activity decay factor	$(0, 1)$
`cla-decay`	double	Clause activity decay factor	$(0, 1)$
`rnd-freq`	double	Frequency for random variable selection	$[0, 1]$
`rnd-init`	bool	Randomize initial activities	{true, false}
`rfirst`	int	Base restart interval	$[1, 1e4]$
`rinc`	double	Restart interval increase factor	$(1.5, 4)$
`gc-frac`	double	Wasted memory fraction triggering garbage collection	$(0, 1)$
`min-learnts`	int	Minimum learnt clause limit	$[0, 1e6]$

Table 8. Kissat configuration parameters

Parameter	Type	Description	Search Space
`chrono`	bool	Enable chronological backtracking	$\{0, 1\}$
`eliminate`	bool	Enable variable elimination	$\{0, 1\}$
`forcephase`	bool	Force initial phase assignment	$\{0, 1\}$
`minimize`	bool	Enable clause minimization	$\{0, 1\}$
`phase`	bool	Set initial decision phase	$\{0, 1\}$
`phasesaving`	bool	Enable phase saving during restarts	$\{0, 1\}$
`probe`	bool	Enable failed literal probing	$\{0, 1\}$
`reduceint`	int	Conflict interval for clause DB reduction	$\{10^1, 10^2, 10^3, 10^4, 10^5\}$
`rephaseint`	int	Conflict interval for phase resetting	$\{10^1, 10^2, 10^3, 10^4, 10^5\}$
`restartint`	int	Base restart interval (conflicts)	$\{1, 10^2, 10^3, 10^4\}$
`restartmargin`	int	Rapid restart margin threshold	$\{0, 5, 10, 15, 20, 25\}$
`simplify`	bool	Enable periodic simplification	$\{0, 1\}$
`stable`	int	Search stability mode (0=focused, 1=stable, 2=switching)	$\{0, 1, 2\}$
`target`	int	Target phase selection strategy (0=negative, 1=positive, 2=best)	$\{0, 1, 2\}$
`tier1`	int	Tier 1 glue limit for learned clauses	$\{2, 3, 4, 5\}$
`tier2`	int	Tier 2 glue limit for learned clauses	$\{6, 7, 8, 9, 10, 20, 50\}$

- **Argumentation problem** involves finding acceptable sets of arguments in a directed graph where attacks between arguments are represented by edges.
- **Cryptography-Ascon problem** represents cryptographic analysis tasks for the lightweight primitive Ascon as SAT instances, where clauses encode round transformations, state relations, and activity constraints to search for valid characteristics or assignments.
- **Social Golfer problem** is a combinatorial problem that aims to assign golfers to groups over several weeks, ensuring no two golfers play in the same group more than once.
- **Hashtable Safety problem** focuses on verifying the correctness of operations in a hash table to avoid collisions and ensure the integrity of the structure.

Table 9. **Solved numbers over training dataset.** This table reports the numbers of solved SAT and UNSAT instances (denoted as "SAT/UNSAT") across multiple benchmark datasets in the training phase. The column *Size* denotes the total number of instances in each dataset. Compared solvers include EasySAT, EasySAT tune, AutoSAT_EA, and AutoSAT_GHC. Higher values in either SAT or UNSAT indicate stronger solving capability.

Dataset	Size	EasySAT	EasySAT tune	AutoSAT_EA	AutoSAT_GHC
argumentation	14	0/0	0/0	1/0	2/0
cryptography-ascon	14	6/8	6/8	6/8	6/8
register-allocation	14	0/0	0/0	0/14	0/0
social-golfer	14	0/0	0/0	6/0	6/0
hashtable-safety	14	0/14	0/14	0/14	0/14
PRP	14	7/0	7/0	7/0	7/0
SCP	23	0/0	0/0	9/0	0/3
brent-equations	13	0/0	0/0	0/1	0/0
mutilated-chessboard	8	0/3	0/3	0/4	0/4
satcoin	10	0/0	0/0	0/10	0/10
grs-fp-comm	11	0/3	0/4	0/4	0/4
school-timetabling	13	11/0	12/0	12/0	12/0
miter	7	0/3	1/3	1/3	1/3
tseitin	7	0/1	0/1	0/1	0/1
CoinsGrid	68	22/0	22/0	27/0	55/0
MineSweeper	61	43/0	42/0	58/0	58/0
KnightTour	39	5/0	7/0	8/0	6/0
LangFord	93	20/0	20/0	27/0	27/0
Zamkeller	56	41/0	41/0	46/0	49/0

- **Register allocation problem** is a problem that arises in compiler optimization, where the goal is to assign a limited number of CPU registers to variables in a program.
- **Profitable-Robust-Production problem (PRP)** focuses on finding a robust production plan that remains profitable under uncertain or fluctuating conditions.
- **Set Covering with Pairs problem (SCP)** is a variant of the classic Set Covering Problem, which is to find a minimum subset of sets from a given collection, such that the union of these sets covers all elements of a given universe. The additional constraint is that certain sets cannot be chosen together.
- **Brent Equations problem** involves finding satisfying assignments to a system of nonlinear arithmetic equations originally proposed by Richard Brent, where the goal is to verify or discover integer solutions under specific constraints.
- **Mutilated Chessboard problem** encode the classic tiling puzzle of covering an 8×8 chessboard with two opposite corners removed using dominoes, requiring SAT reasoning to prove (un)satisfiability of possible tilings.
- **Satcoin problem** represents cryptographic coin-mixing or blockchain transaction–verification tasks as SAT instances, where clauses encode balance conservation and transaction dependencies.

Table 10. **Solved numbers over training dataset.** This table reports the number of solved SAT and UNSAT instances across multiple benchmark datasets in the training phase. The column *Size* denotes the total number of instances in each dataset. Compared solvers include MiniSat, MiniSat tune, Kissat, Kissat tune, and AutoSAT (the better result between AutoSAT_EA and AutoSAT_GHC). For each dataset, the results are reported in the format "SAT/UNSAT", where higher values indicate stronger solving capability.

Dataset	Size	MiniSat	MiniSat tune	Kissat	Kissat tune	AutoSAT
argumentation	14	0/2	1/2	1/8	1/8	2/0
cryptography-ascon	14	6/8	6/8	6/8	6/8	6/8
register-allocation	14	0/0	0/0	0/4	0/7	0/14
social-golfer	14	1/0	1/0	2/0	5/0	6/0
hashtable-safety	14	0/14	0/14	0/14	0/14	0/14
PRP	14	7/0	7/0	7/0	7/0	7/0
SCP	23	0/9	0/9	14/9	14/9	9/0
brent-equations	13	0/1	0/1	11/1	11/1	0/1
mutilated-chessboard	8	0/0	0/0	0/6	0/6	0/4
satcoin	10	0/0	0/0	0/10	0/10	0/10
grs-fp-comm	11	7/4	7/4	0/8	0/8	0/4
school-timetabling	13	8/0	8/0	11/0	12/0	12/0
miter	7	1/3	1/3	1/5	1/5	1/3
tseitin	7	0/1	0/1	0/1	0/1	0/1
CoinsGrid	68	31/0	31/0	43/0	47/0	55/0
MineSweeper	61	61/0	61/0	61/0	61/0	58/0
KnightTour	39	4/0	6/0	6/0	6/0	8/0
LangFord	93	15/0	14/0	49/0	49/0	27/0
Zamkeller	56	28/0	31/0	49/0	55/0	49/0

- **GRS-FP-Comm problem** models fault-propagation and communication constraints in grid resource scheduling (GRS) with floating-point conditions, asking for a schedule that satisfies all timing and communication requirements.
- **Family School Timetabling problem** requires constructing conflict-free weekly schedules for families and schools, ensuring that shared resources (teachers, rooms, or students) do not overlap and that all curricular constraints are met.
- **Miter problem** is a hardware-verification benchmark where a miter circuit is built to compare two designs; the SAT solver checks whether any input assignment causes the two circuits to produce different outputs.
- **Tseitin problem** generates SAT formulas from graphs using the Tseitin transformation, creating hard unsatisfiable instances by assigning parity constraints to graph vertices and encoding them into conjunctive normal form.
- **CoinsGrid problem** is derived from Tony Hurlimann's coin puzzle, focusing on arranging coins on a grid with specific row, column, and distance constraints.

Table 11. **Solved numbers over testing dataset.** This table reports the number of solved SAT and UNSAT instances (denoted as "SAT/UNSAT") across multiple benchmark datasets in the testing phase. The column *Size* denotes the total number of instances in each dataset. Compared solvers include EasySAT, EasySAT tune, AutoSAT_EA and AutoSAT_GHC. Higher values in either SAT or UNSAT indicate stronger solving capability.

Dataset	Size	EasySAT	EasySAT tune	AutoSAT_EA	AutoSAT_GHC
argumentation	6	0/0	0/0	1/0	0/0
cryptography-ascon	6	2/4	2/4	2/4	2/4
register-allocation	6	0/0	0/0	0/5	0/0
social-golfer	6	0/0	0/0	3/0	3/0
hashtable-safety	6	0/6	0/6	0/6	0/6
PRP	6	3/0	3/0	3/0	3/0
SCP	10	0/0	0/0	5/0	0/2
brent-equations	6	0/0	0/0	0/0	0/0
mutilated-chessboard	4	0/0	0/0	0/0	0/1
satcoin	5	0/0	0/0	0/5	0/5
grs-fp-comm	6	0/3	0/3	0/3	0/3
school-timetabling	6	5/0	5/0	6/0	5/0
miter	4	0/2	0/2	0/2	0/2
tseitin	3	0/0	0/0	0/0	0/0
CoinsGrid	30	11/0	11/0	12/0	13/0
MineSweeper	27	23/0	23/0	25/0	25/0
KnightTour	17	1/0	2/0	2/0	2/0
LangFord	41	7/0	7/0	10/0	10/0
Zamkeller	24	13/0	14/0	15/0	17/0

- **MineSweeper problem** is derived from the classic MineSweeper game, where the objective is to determine the placement of hidden mines on a grid based on numerical clues.
- **LangFord problem** is a combinatorial mathematics problem that involves finding a specific permutation of the sequence 1, 1, 2, 2, ..., n, n where the two copies of each number k are exactly k units apart.
- **KnightTour problem** aims to find a path for a knight on a chessboard that visits every square exactly once, with possible extensions to different board sizes and types.
- **Zamkeller problem** involves finding a permutation of integers from 1 to n that maximizes the number of differential alternations in subsequences divisible by integers from 1 to k, where $(1 < k < n)$.

B.5 Problem Generation

In this section, we provide details on the generated data based on Picat (Zhou et al. 2015). Specifically, the scalable problems following the settings in Chapters 2 and 3 in the book (Zhou et al. 2015) are adopted. Sampling these instances requires specific parameter settings, and we conducted grid sampling within the parameter space presented below. A parameter space $\Theta = \{\theta_1, \theta_2, \ldots\}$ is defined with $\theta_i^L \leq \theta_i \leq \theta_i^U$, ensuring that the three baseline solvers EasySAT, MiniSat and Kissat can obtain a solution within reasonable CPU time, i.e., [1s, 5000s].

Table 12. **Solved numbers over testing dataset.** This table reports the number of solved SAT and UNSAT instances (denoted as "SAT/UNSAT") across multiple benchmark datasets in the testing phase. The column *Size* denotes the total number of instances in each dataset. Compared solvers include MiniSat, MiniSat tune, Kissat, Kissat tune, and AutoSAT (best of AutoSAT_EA and AutoSAT_GHC). Higher values in either SAT or UNSAT indicate stronger solving capability.

Dataset	Size	MiniSat	MiniSat tune	Kissat	Kissat tune	AutoSAT
argumentation	6	1/0	0/0	1/4	1/4	1/0
cryptography-ascon	6	2/4	2/4	2/4	2/4	2/4
register-allocation	6	0/0	0/0	0/1	0/1	0/5
social-golfer	6	3/0	1/0	3/0	3/0	3/0
hashtable-safety	6	0/6	0/6	0/6	0/6	0/6
PRP	6	3/0	3/0	3/0	3/0	3/0
SCP	10	4/0	0/4	6/4	6/4	5/0
brent-equations	6	0/0	0/0	5/1	5/0	0/0
mutilated-chessboard	4	0/1	0/0	0/3	0/3	0/1
satcoin	5	0/0	0/0	0/5	0/5	0/5
grs-fp-comm	6	0/3	0/3	0/5	0/5	0/3
school-timetabling	6	5/0	5/0	5/0	5/0	6/0
miter	4	0/2	0/2	0/2	0/2	0/2
tseitin	3	0/0	0/0	0/0	0/0	0/0
CoinsGrid	30	12/0	12/0	19/0	23/0	13/0
MineSweeper	27	27/0	27/0	27/0	27/0	25/0
KnightTour	17	1/0	2/0	1/0	2/0	2/0
LangFord	41	4/0	5/0	16/0	16/0	10/0
Zamkeller	24	8/0	9/0	17/0	24/0	17/0

In addition, an extended space Θ' is introduced by enlarging the upper bound of Θ, e.g., $\theta_i'^U = \theta_i^U \times 1.2$, which allows us to test whether AutoSAT can solve instances that the baseline solvers cannot.

- **CoinsGrid problem**
  Parameter Θ: $\{n, c\}$
  Parameter Space: $[16, 157] \times [45, 400]$
  Notes: An $n \times n$ grid with c coins in each row and each column.
- **MineSweeper problem**
  Parameter Θ: $\{m, n, k, p\}$
  Parameter Space: $[500, 1600] \times [400, 3200] \times [72689, 1572118] \times [0.32, 0.38]$
  Notes: m, n represent the grid size of the Minesweeper game. k is the total number of mines. p is the probability that a given cell contains a mine (range: 0.32 to 0.38).
- **LangFord problem**
  Parameter Θ: $\{k\}$
  Dataset Range: $[50, 291]$

Notes: A sequence of $2k$ elements, where numbers from 1 to k appear twice. The first occurrence of a number i is exactly i positions away from the second occurrence.

- **KnightTour problem**
  Parameter Θ: $\{k\}$
  Parameter Space: $[12, 75]$
  Notes: A $k \times k$ chessboard where a knight's tour is attempted, covering all squares and returning to the start point. (A solution is not possible for odd-sized boards, i.e., they are unsatisfiable.)
- **Zamkeller problem**:
  Parameter Θ: $\{k, n\}$
  Parameter Space: $[3, 34] \times [25, 100]$
  Notes: n represents the total sequence length, and k represents the subsequence length. For all subsequences of length k, the goal is to change them into the minimum number of distinct sequences.

C Prompt Template and Answers

This appendix summarizes the prompt templates and representative responses used throughout our study. Section C.1 presents several motivation questions used to assess whether LLMs have the basic knowledge needed for SAT solving. Section C.2 reports a preliminary attempt to generate a competitive SAT solver from scratch, which further motivates our choice to refine an existing solver instead of constructing one directly. Section C.3 gives the prompt templates adopted in our experiments.

C.1 Prompts and Answers in Motivation Questions

To test whether LLMs have basic information about SAT solvers that can enable them to improve the solvers. We ask LLMs the following questions:

- Can you tell me the definition of SAT?
- Can you provide the pseudocode of a CDCL solver?
- Can you write C++ code based on the pseudocode?

It was found that LLMs could successfully provide details of SAT and generate a basic CDCL framework. However, due to the limitations in the text length, the generated code tends to be too brief and simplified to meet the standards of modern SAT solvers. Therefore, we opted to refine an existing solver rather than construct one from scratch. The LLMs' responses are presented below.

Answer of Question 1: Can you tell me the definition of SAT?

```
Yes, I understand the Boolean Satisfiability Problem (often abbreviated as SAT). It's a
    decision problem that asks whether there exists an assignment of truth values to
    variables that makes a given Boolean formula true. More formally, the SAT problem
    involves determining if there is a way to assign True or False to variables in a Boolean
    formula (typically expressed in Conjunctive Normal Form, or CNF) so that the entire
    formula evaluates to True.

Example:
Consider the Boolean formula:

This formula is satisfiable if there is an assignment of True and False to x_1 and x_2 that
    makes the entire formula evaluate to True.
One possible satisfying assignment is x_1 = True and x_2 = True.
```

```
CNF (Conjunctive Normal Form):
SAT problems are often expressed in CNF, which is a conjunction (AND) of clauses, where each
    clause is a disjunction (OR) of literals (a variable or its negation). A formula is
    satisfiable if there is some way to make each clause in the CNF true.
```

Answer of Question 2: Can you provide the pseudocode of CDCL solver?

```
CDCL_SAT_Solver(CNF_formula):
    initialize assignment_stack = []
    initialize clause_database = CNF_formula
    initialize decision_level = 0

    while True:
        # Perform unit propagation to make implied assignments
        conflict_clause = Unit_Propagation(clause_database, assignment_stack)

        if conflict_clause is not None:
            # Conflict occurred, analyze and learn a new clause
            if decision_level == 0:
                # If at decision level 0, unsatisfiable
                return UNSAT
            else:
                # Analyze the conflict and learn a new clause
                learned_clause, backtrack_level = Analyze_Conflict(conflict_clause)

                # Add the learned clause to the clause database
                clause_database.add(learned_clause)

                # Backjump to the level indicated by conflict analysis
                Backjump(assignment_stack, backtrack_level)
                decision_level = backtrack_level

        else:
            # No conflict, check if all variables are assigned
            if All_Variables_Assigned(assignment_stack):
                return SAT  # Satisfiable solution found

            # Make a decision: choose a new variable and assign it
            decision_literal = Pick_Unassigned_Variable(assignment_stack)
            decision_level += 1
            assignment_stack.push(decision_literal)

            # Optionally restart the search based on a heuristic
            if Should_Restart():
                Restart_Search(assignment_stack)
                decision_level = 0

# Helper Functions:

Unit_Propagation(clause_database, assignment_stack):
    while there are unit clauses in clause_database:
```

```
        assign the unit clause's remaining literal to satisfy the clause
        if a conflict arises:
            return conflict_clause
    return None  # No conflict

Analyze_Conflict(conflict_clause):
    learned_clause = derive a new clause from the conflict (e.g., using resolution)
    backtrack_level = determine the level to backtrack to
    return learned_clause, backtrack_level

Backjump(assignment_stack, backtrack_level):
    remove all assignments in assignment_stack after backtrack_level
    restore the state of the solver at that level

Pick_Unassigned_Variable(assignment_stack):
    choose the next variable to assign (often based on a heuristic like VSIDS or activity-
     based scoring)
    return decision_literal

Should_Restart():
    return True if the solver should restart (based on a restart policy), otherwise False

Restart_Search(assignment_stack):
    clear the assignment_stack but retain learned clauses in clause_database
```

Answer of Question 3: Can you write C++ code based on the pseudocode?

```
#include <iostream>
#include <vector>
#include <stack>
#include <set>
#include <unordered_map>

using namespace std;

// Represents a clause, which is a disjunction of literals.
typedef vector<Literal> Clause;

class CDCLSolver {
public:
    CDCLSolver(const vector<Clause>& formula) : clauseDatabase(formula), decisionLevel(0) {}

    // Runs the CDCL algorithm on the CNF formula.
    bool solve() {
        while (true) {
            Clause conflictClause = unitPropagation();
            if (!conflictClause.empty()) {
                // Conflict found
                if (decisionLevel == 0) {
                    return false; // UNSAT
                } else {
```

```
                    Clause learnedClause;
                    int backtrackLevel;
                    tie(learnedClause, backtrackLevel) = analyzeConflict(conflictClause);
                    clauseDatabase.push_back(learnedClause);
                    backjump(backtrackLevel);
                    decisionLevel = backtrackLevel;
                }
            } else {
                if (allVariablesAssigned()) {
                    return true; // SAT
                }
                // Make a decision
                Literal decisionLiteral = pickUnassignedVariable();
                decisionStack.push(decisionLiteral);
                decisionLevel++;
                assignment[abs(decisionLiteral)] = (decisionLiteral > 0);
            }

            if (shouldRestart()) {
                restartSearch();
            }
        }
    }

private:
    vector<Clause> clauseDatabase; // CNF formula
    stack<Literal> decisionStack;  // Stack of decisions
    unordered_map<int, bool> assignment;  // Maps variables to their assigned truth values
    int decisionLevel;  // Current decision level in the search

    // Helper function: Perform unit propagation
    Clause unitPropagation() {
        // Simplified unit propagation
        while (true) {
            bool conflict = false;
            for (const Clause& clause : clauseDatabase) {
                int unassignedCount = 0;
                Literal unassignedLiteral = 0;
                bool clauseSatisfied = false;
                for (Literal lit : clause) {
                    int var = abs(lit);
                    if (assignment.find(var) != assignment.end()) {
                        if ((lit > 0 && assignment[var]) || (lit < 0 && !assignment[var])) {
                            clauseSatisfied = true;
                            break;
                        }
                    } else {
                        unassignedCount++;
                        unassignedLiteral = lit;
                    }
                }
                if (!clauseSatisfied && unassignedCount == 0) {
                    return clause; // Conflict found
                } else if (!clauseSatisfied && unassignedCount == 1) {
```

```
                // Unit clause, assign the literal
                assignment[abs(unassignedLiteral)] = (unassignedLiteral > 0);
                decisionStack.push(unassignedLiteral);
            }
        }
        if (!conflict) break;
    }
    return {}; // No conflict
}

// Helper function: Analyze conflict and learn a new clause
pair<Clause, int> analyzeConflict(const Clause& conflictClause) {
    Clause learnedClause = conflictClause;  // Simplified conflict analysis (can be
 improved)
    int backtrackLevel = decisionLevel - 1; // Backtrack one level for simplicity
    return {learnedClause, backtrackLevel};
}

// Helper function: Perform backjumping to a specific decision level
void backjump(int backtrackLevel) {
    while (!decisionStack.empty() && decisionLevel > backtrackLevel) {
        Literal lastDecision = decisionStack.top();
        decisionStack.pop();
        assignment.erase(abs(lastDecision));  // Unassign variables
        decisionLevel--;
    }
}

// Helper function: Pick an unassigned variable for decision
Literal pickUnassignedVariable() {
    for (int i = 1; i <= 10; ++i) {  // Assume variables are from 1 to 10 for simplicity
        if (assignment.find(i) == assignment.end()) {
            return i;  // Return the first unassigned variable
        }
    }
    return 0;  // All variables are assigned
}

// Helper function: Check if all variables are assigned
bool allVariablesAssigned() {
    for (int i = 1; i <= 10; ++i) {  // Assume variables are from 1 to 10 for simplicity
        if (assignment.find(i) == assignment.end()) {
            return false;
        }
    }
    return true;
}

// Helper function: Determine if a restart is necessary
bool shouldRestart() {
    // Simplified: Always return false (can implement heuristics here)
    return false;
}
```

```
    // Helper function: Restart the search
    void restartSearch() {
        // Reset the assignment and decision stack
        decisionStack = stack<Literal>();
        assignment.clear();
        decisionLevel = 0;
    }
};

int main() {
    vector<Clause> cnfFormula = {{1, 2}, {-1, 3}, {-2, -3}};

    CDCLSolver solver(cnfFormula);
    bool result = solver.solve();

    if (result) cout << "SATISFIABLE" << endl;
    else cout << "UNSATISFIABLE" << endl;

    return 0;
}
```

C.2 Experiments of Generating Solver Code from Scratch

The motivation questions shown before raises the question of whether LLMs could generate a competitive SAT solver directly from scratch. To examine this possibility, we follow the approach of FunSearch (Romera-Paredes et al. 2024) and attempt to generate complete SAT solver code directly using LLMs. The prompt template is based on EoH (F. Liu et al. 2024). It turns out LLMs only generate code within 1k tokens, which is far from sufficient to achieve the performance of commonly used SAT solvers. Note that the codebase of typical SAT solvers is quite large; even EasySAT, the most lightweight CDCL solver, exceeds 5k tokens.

The generated solver is based on a simple CDCL algorithm without complex data structures or heuristics. When evaluated on our benchmark, it fails to solve any instances; therefore, we do not report the results.

EoH Prompt

I need help designing a SAT solver that efficiently solves Boolean satisfiability problems. The solver should be able to process a CNF formula and return a satisfying assignment if one exists or indicate that the formula is unsatisfiable. The final goal is to optimize the solver's runtime performance and accuracy.

Please design a new SAT-solving algorithm.
Firstly, describe your new algorithm and its main steps in one sentence. Next, implement it in C++ as a function named solve_sat. This function should accept one input: cnf_formula. The function should return one output: result. cnf_formula is a list of clauses, where each clause is a list of literals represented as integers. The output named result should be a tuple containing a Boolean value indicating satisfiability and a dictionary representing a satisfying assignment if one exists, or an empty dictionary if unsatisfiable.

Note that cnf_formula is a list of lists with integer values indicating literals, and the function should not involve any probabilistic components. Ensure that the algorithm maintains self-consistency and does not provide additional explanations.

Answer Code 1:

```
#include <vector>
#include <unordered_map>
#include <optional>

bool solve_dpll(const std::vector<std::vector<int>>& cnf_formula, std::unordered_map<int,
    bool>& assignment) {
    if (cnf_formula.empty()) return true;

    for (const auto& clause : cnf_formula) {
        if (clause.empty()) return false;
    }

    // Choose first unassigned variable (naive implementation)
    int variable = 0;
    for (const auto& clause : cnf_formula) {
        for (int literal : clause) {
            variable = abs(literal);
            if (assignment.find(variable) == assignment.end()) goto found_variable;
        }
    }
found_variable:

    for (int value : {1, -1}) {
        assignment[variable] = (value == 1);

        // Propose assignment with current variable choice
        std::vector<std::vector<int>> new_cnf_formula;
        for (const auto& clause : cnf_formula) {
            bool clause_is_satisfied = false;
            std::vector<int> new_clause;
            for (int literal : clause) {
                if (literal == variable && value == 1) {
                    clause_is_satisfied = true;
                    break;
                } else if (literal == -variable && value == -1) {
                    clause_is_satisfied = true;
                    break;
                } else if (literal != variable && literal != -variable) {
                    new_clause.push_back(literal);
                }
            }
            if (!clause_is_satisfied) new_cnf_formula.push_back(new_clause);
        }

        if (solve_dpll(new_cnf_formula, assignment)) return true;

        assignment.erase(variable);
    }
    return false;
}
```

```
std::pair<bool, std::unordered_map<int, bool>> solve_sat(const std::vector<std::vector<int>>&
     cnf_formula) {
    std::unordered_map<int, bool> assignment;
    bool is_satisfiable = solve_dpll(cnf_formula, assignment);
    return {is_satisfiable, is_satisfiable ? assignment : std::unordered_map<int, bool>{}};
}
```

Answer Code 2:

```
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <iostream>

bool solve_recursive(std::vector<std::vector<int>>& cnf_formula, std::unordered_map<int, bool
    >& assignment) {
    // Check if all clauses are satisfied
    bool all_satisfied = true;
    for (const auto& clause : cnf_formula) {
        bool clause_satisfied = false;
        for (int literal : clause) {
            int var = abs(literal);
            bool value = (literal > 0);

            if (assignment.find(var) != assignment.end() && assignment[var] == value) {
                clause_satisfied = true;
                break;
            }
        }
        if (!clause_satisfied) {
            all_satisfied = false;
            break;
        }
    }
    if (all_satisfied) return true;
    // Check if a clause is unsatisfied
    for (const auto& clause : cnf_formula) {
        bool clause_unsatisfied = true;
        for (int literal : clause) {
            int var = abs(literal);
            if (assignment.find(var) == assignment.end()) {
                clause_unsatisfied = false;
                break;
            }
        }
        if (clause_unsatisfied) return false;
    }

    // Choose an unassigned variable
    int variable = 0;
    for (const auto& clause : cnf_formula) {
```

```
        for (int literal : clause) {
            int var = abs(literal);
            if (assignment.find(var) == assignment.end()) {
                variable = var;
                break;
            }
        }
        if (variable != 0) break;
    }

    assignment[variable] = true;
    if (solve_recursive(cnf_formula, assignment)) return true;
    assignment[variable] = false;
    if (solve_recursive(cnf_formula, assignment)) return true;
    assignment.erase(variable);
    return false;
}

std::pair<bool, std::unordered_map<int, bool>> solve_sat(std::vector<std::vector<int>>&
    cnf_formula) {
    std::unordered_map<int, bool> assignment;
    bool is_satisfiable = solve_recursive(cnf_formula, assignment);
    return { is_satisfiable, is_satisfiable ? assignment : std::unordered_map<int, bool>() };
}

int main() {
    std::vector<std::vector<int>> cnf_formula = { {1, -3, 4}, {-1, 2, 3}, {-2, -3}, {2, -4}
    };
    auto result = solve_sat(cnf_formula);
    std::cout << "Satisfiable: " << result.first << std::endl;
    if (result.first) {
        std::cout << "Assignment: ";
        for (const auto& assign : result.second) {
            std::cout << assign.first << "=" << assign.second << " ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

Answer Code 3:

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <optional>
using namespace std;

struct SATResult {
    bool satisfiable;
    unordered_map<int, bool> assignment;
```

```
};

optional<unordered_map<int, bool>> DPLL(const vector<vector<int>>& cnf_formula, unordered_map
    <int, bool>& assignment) {
    for (const auto& clause : cnf_formula) {
        bool clause_satisfied = false;
        bool has_unassigned_literal = false;
        for (int literal : clause) {
            auto it = assignment.find(abs(literal));
            if (it != assignment.end() && it->second == (literal > 0)) {
                clause_satisfied = true;
                break;
            }
            if (it == assignment.end()) {
                has_unassigned_literal = true;
            }
        }
        if (!clause_satisfied && !has_unassigned_literal) {
            return {}; // Clause unsatisfiable and no unassigned literal
        }
    }

    int unassigned_var = 0;
    for (const auto& clause : cnf_formula) {
        for (int literal : clause) {
            if (assignment.find(abs(literal)) == assignment.end()) {
                unassigned_var = abs(literal);
                break;
            }
        }
        if (unassigned_var != 0) break;
    }

    if (unassigned_var == 0)  return assignment;

    assignment[unassigned_var] = true;
    auto result = DPLL(cnf_formula, assignment);
    if (result.has_value()) {
        return result;
    }
    assignment[unassigned_var] = false;
    return DPLL(cnf_formula, assignment);
}

SATResult solve_sat(const vector<vector<int>>& cnf_formula) {
    unordered_map<int, bool> assignment;
    auto result = DPLL(cnf_formula, assignment);
    if (result.has_value()) {
        return {true, result.value()};
    }
    return {false, {}};
}

int main() {
```

```
    vector<vector<int>> cnf_formula = {{1, -3}, {-1, 2, 3}, {-1, -2}};
    SATResult result = solve_sat(cnf_formula);

    cout << "Satisfiable: " << result.satisfiable << endl;
    if (result.satisfiable) {
        cout << "Assignment: ";
        for (const auto& [var, val] : result.assignment) {
            cout << var << ": " << val << ", ";
        }
        cout << endl;
    }
    return 0;
}
```

C.3 Prompts and Answers in Experiments

The success of AutoSAT heavily depends on prompt engineering. The prompt templates for the three LLM agents are presented below.

Advisor Prompt

Your goal is to write a bump_var function for the SAT solver that will help it restart the search and escape from the local optimum, after reading and understanding the <key code> of SAT solver below.
Your answer must follow the following format:
Description: ...
Modification Direction:

Tips:
1) You must traverse all possible position of <key code> if you want to modify the {{ task }}
2) You need to give us some advice to modify the function. e.g. some potential directions to change the heuristics. (The directions should be feasible)
3) Notice that, you can only change this function.

<key code> of SAT solver is:
...

Advisor Answer:

```
Description: The bump\_var function is a critical part of the SAT solver's variable activity
    management, which is used to escape from local optima by dynamically adjusting the
    activity scores of variables. The activity score of a variable is increased by a certain
    coefficient whenever it is involved in a conflict. If the activity score becomes too
    large, the scores of all variables are scaled down to prevent overflow. The function also
     updates the variable's position in the VSIDS (Variable State Independent Decaying Sum)
    heap if it is already present. Key variables include 'var' (the variable to be bumped), '
    coeff' (the coefficient by which the activity is increased), 'activity' (an array storing
     the activity scores of variables), and 'var\_inc' (a variable increment factor). The
    bump\_var function is called in the analyze function to adjust the activity scores of
    variables involved in conflicts and in other parts of the solver where heuristic
    decisions are made.

Modification Direction:
    (1) Introduce a dynamic coefficient adjustment mechanism based on the frequency of
    conflicts to make the bumping more adaptive.
    (2) Implement a decay mechanism that decreases the activity of all variables periodically,
     not just when overflow is imminent, to better balance exploration and exploitation.
    (3) Incorporate machine learning techniques to predict the most promising variables to
    bump based on historical data and patterns observed during the solving process.
    (4) Adjust the bumping strategy based on the type of conflicts (e.g., recent conflicts vs.
     historical conflicts) to prioritize variables that are more likely to lead to a solution
    .
    (5) Experiment with different forms of variable activity updates, such as exponential or
    logarithmic scaling, to see if they lead to better performance in escaping local optima.
```

Coder Prompt

You are a SAT solver researcher trying to rewrite the bump_var function(s).

Your goal is to improve the SAT solver by rewriting the bump_var function(s), after reading and understanding the <key code> of SAT solver below.

Tips:
1) Your rewritten function code must start with "'// start function name"' and end with "'// end function name"'
2) Your rewritten function(s) code must be different from original code, not just a rewrite code synonymous!
3) You are not allowed to create your own new function(s) in the rewritten function(s). You are not allowed to create your own new global variables, but you can use the global variables existing in the <key code>.
4) Make sure the rewritten function(s) code can be executed correctly.

Advice of modifying this function is:
...
<key code> of SAT solver is:
...

Coder Answer:

Based on the provided key code of the SAT solver, I will rewrite the `bump_var` function to improve the SAT solver. The goal is to modify the function in a way that is different from the original code, without creating new functions or global variables, and ensuring that the rewritten function can be executed correctly.

Here is the rewritten `bump_var` function:

```cpp
// start bump_var
void Solver::bump_var(int var, double coeff) {
    activity[var] += var_inc * coeff; // Increment activity by the variable increment times the coefficient

    // Check if the activity of any variable exceeds a large threshold, and normalize if so.
    const double large_threshold = 1e100;
    bool normalize = false;
    for (int i = 1; i <= vars; i++) {
        if (activity[i] > large_threshold) {
            normalize = true;
            break;
        }
    }
    if (normalize) {
        double max_activity = 0;
        for (int i = 1; i <= vars; i++) {
            if (activity[i] > max_activity) {
                max_activity = activity[i];
            }
        }
        for (int i = 1; i <= vars; i++) {
            activity[i] /= max_activity;
        }
        var_inc /= max_activity;
    }

    // Update the variable's position in the heap if it is present
    if (vsids.inHeap(var)) vsids.update(var);
}
// end bump_var
```

In this rewritten version, I've removed the periodic normalization that was previously done every 5000 conflicts. Instead, I've introduced a check that normalizes the activities if any variable's activity exceeds a large threshold (1e100). This approach avoids the need for a periodic check and focuses on the condition that actually necessitates normalization. The normalization process itself is also slightly modified to iterate through the activities twice, first to find the maximum activity and then to normalize, which may be more efficient if normalization is a rare event.

Evaluator Prompt

Your goal is to evaluate whether the <function you wrote> of the SAT solver is appropriate, after reading and understanding the <key code> of SAT solver below. Then, you need to inform me about the modifications to the bump_var function are "Substantial Improvement," "Parameter Tuning," or "No Modification".

Your answer must follow the following format:
Type: choose one from “Substantial Improvement”, “Parameter Tuning”, “No Modification”,
Extra_analysis: Provide a logical and concise analysis of the bump_var function you wrote and explain its underlying intuition compared with original code.

Tips:
1) You must check for syntax errors to ensure it can run correctly.
2) You need to compare the bump_var function you modified to the original code and determine if there are substantial improvements (note that parameter adjustments alone do not count).
3) Sometimes, although there are differences between the modified code and the original code, these are just changes to variable names or synonymous transformations in statements. Such cases should also be considered as parameter tuning.
4) If there are substantial modifications to the revamped code, respond with: “Substantial Improvement”; If there are no substantial changes, merely adjustments to parameters, respond with: “Parameter Tuning”; If there are no modifications at all including adding intermediate parameters, respond with: “No Modification”.
5) If you think the “type” is “No modification”, then “extra_analysis” should be empty.
6) “extra_analysis” should be brief and no more than 3 sentences.

<bump_var function you wrote>:
...
<key code> of SAT solver is:
...

Evaluator Answer:

```
Type: ``Substantial Improvement''
Extra_analysis: The modified `bump_var` introduces a dynamic coefficient adjustment mechanism
     based on the frequency of conflicts, which is a substantial change from the original
    static coefficient. This dynamic adjustment aims to better adapt the activity score
    updates to the solver's current state, potentially improving its ability to escape local
    optima.
```

D Code of Discovered Heuristics

To demonstrate LLMs’ ability to generate effective heuristics, we provide contrastive code examples (one for each function candidate) in this section. These materials supplement Section 4.3 with concrete implementations of the discovered heuristics.

original reduce_function

```
void Solver::reduce() {
    backtrack(0);
    reduces = 0, reduce_limit += 512;
    int new_size = origin_clauses, old_size = clause_DB.size();
    reduce_map.resize(old_size);
    for (int i = origin_clauses; i < old_size; i++) {    // random delete 50% bad clauses (LBD>=5)
        if (clause_DB[i].lbd >= 5 && rand() % 2 == 0) reduce_map[i] = -1;  // remove clause
        else {
            if (new_size != i) clause_DB[new_size] = clause_DB[i];
            reduce_map[i] = new_size++;
        }
    }
    clause_DB.resize(new_size, Clause(0));
    for (int v = -vars; v <= vars; v++) {   // Update the watches.
        if (v == 0) continue;
        int old_sz = watch(v).size(), new_sz = 0;
        for (int i = 0; i < old_sz; i++) {
            int old_idx = watch(v)[i].idx_clause;
            int new_idx = old_idx < origin_clauses ? old_idx : reduce_map[old_idx];
            if (new_idx != -1) {
                watch(v)[i].idx_clause = new_idx;
                if (new_sz != i) watch(v)[new_sz] = watch(v)[i];
                new_sz++;
            }
        }
        watch(v).resize(new_sz);
    }
}
```

Fig. 2. Example of original heuristics of reduce function

discovered reduce_function

```
void Solver::reduce() {
    backtrack(0);
    reduces = 0;
    reduce_limit = reduce_limit * 1.1 + 512;  // Adaptive increase
    int new_size = origin_clauses;
    int old_size = clause_DB.size();
    reduce_map.resize(old_size);
    // Two-pass strategy: aggressive remove high-LBD clauses, conserve low-LBD clauses
    for (int i = origin_clauses; i < old_size; i++) {
        Clause& clause = clause_DB[i];
        double removal_probability = 0.0;
        // Dynamic removal probability based on LBD, age, and activity
        if (clause.lbd >= 8) removal_probability = 0.8;  // Aggressive removal
        else if (clause.lbd >= 5) removal_probability = 0.5;  // Moderate removal
        else if (clause.lbd >= 3) removal_probability = 0.2;  // Conservative removal
        // Adjust probability based on clause age
        double age_factor = 1.0 - exp(-(old_size - i) / 1000.0);
        removal_probability *= (0.3 + 0.7 * age_factor);
        // Random removal with calculated probability
        if (((double)rand() / RAND_MAX) < removal_probability) {
            reduce_map[i] = -1;  // Mark for removal
        } else {
            if (new_size != i) {
                clause_DB[new_size] = std::move(clause);
            }
            reduce_map[i] = new_size++;
        }
    }
    clause_DB.resize(new_size, Clause(0));
    for (int v = -vars; v <= vars; v++) {   // Update the watches.
        if (v == 0) continue;
        int old_sz = watch(v).size(), new_sz = 0;
        for (int i = 0; i < old_sz; i++) {
            int old_idx = watch(v)[i].idx_clause;
            int new_idx = old_idx < origin_clauses ? old_idx : reduce_map[old_idx];
            if (new_idx != -1) {
                watch(v)[i].idx_clause = new_idx;
                if (new_sz != i) watch(v)[new_sz] = watch(v)[i];
                new_sz++;
            }
        }
        watch(v).resize(new_sz);
    }
}
```

Fig. 3. Example of LLM discovered heuristics of reduce function

original rephase_function

```
void Solver::rephase() {
    rephases = 0, threshold *= 0.9, rephase_limit += 8192;
}
```

discovered rephase_function

```
void Solver::rephase() {
    rephases = 0;
    rephase_limit = rephase_limit * 1.3 + 2048;  // More adaptive increase

    // Dynamic phase selection based on solver state
    double progress = (double)conflicts / (conflicts + 10000);
    int strategy = rand() \% 100;

    if (strategy < 60 - progress * 30) {  // Progressive shift from local to activity-based
        // Activity-weighted phase selection with momentum
        for (int i = 1; i <= vars; i++) {
            if (activity[i] > var_inc * 100) {
                double activity_ratio = log(1 + activity[i] / var_inc) / 10;
                saved[i] = (rand() \% 100 < (50 + activity_ratio * 40)) ? 1 : -1;
            } else {
                saved[i] = local_best[i];
            }
        }
    }
    else if (strategy < 85) {  // Local best with occasional exploration
        for (int i = 1; i <= vars; i++) {
            if (rand() \% 100 < 15) {
                saved[i] = (rand() \% 2) ? 1 : -1;
            } else {
                saved[i] = local_best[i];
            }
        }
    }
    else {  // Strong diversification
        for (int i = 1; i <= vars; i++) {
            saved[i] = (local_best[i] != 0) ? -local_best[i] : (rand() \% 2 ? 1 : -1);
        }
    }

    // Update threshold based on recent search progress
    threshold = trail.size() * (0.7 + 0.3 * progress);
}
```

Fig. 4. Example of LLM discovered heuristics of rephase function

original heuristic_bump_var

```
void Solver::heuristic_bump_var(int backtrackLevel, std::vector<int> &bump)
{
     for (int i = 0; i < (int)bump.size(); i++)       // heuristically bump some variables.
        if (level[bump[i]] >= backtrackLevel - 1) {
           bump_var(bump[i], 1);
        }
}
```

discoverd heuristic_bump_var

```
void Solver::heuristic_bump_var(int backtrackLevel, std::vector<int> &bump)
{
    double decay_factor = 0.95;
    double level_bonus = 1.2;

    for (int i = 0; i < (int)bump.size(); i++) {
        int var = bump[i];
        if (level[var] >= backtrackLevel) {
            // Give higher bonus to variables at or above backtrack level
            double bonus = (level[var] >= backtrackLevel) ? level_bonus : 1.0;
            // Apply exponential decay based on how many times variable has been bumped
            double decay = pow(decay_factor, mark[var]);
            bump_var(var, bonus * decay);
            mark[var]++; // Track bump count for decay calculation
        }
    }
}
```

Fig. 5. Example of LLM discovered heuristics of varBumpActivity

original restart_condition

```
bool Solver::restart_condition()
{
    if (lbd_queue_size == 50 && 0.8 * fast_lbd_sum / lbd_queue_size > slow_lbd_sum / conflicts)
        return true;
    else return false;
}
```

discoverd restart_condition

```
bool Solver::restart_condition() {
    double fast_avg = fast_lbd_sum / lbd_queue_size;
    double slow_avg = slow_lbd_sum / conflicts;
    double threshold_factor = 0.65;  // Adjusted threshold factor for more sensitivity

    // Introduce a dynamic threshold based on the number of conflicts and LBD variance
    double conflict_rate = static_cast<double>(conflicts) / (conflicts + 1);
    double lbd_variance = 0.0;
    for (int i = 0; i < lbd_queue_size; i++) {
        double diff = lbd_queue[i] - fast_avg;
        lbd_variance += diff * diff;
    }
    lbd_variance /= lbd_queue_size;

    double dynamic_threshold = 1.0 + (conflict_rate * lbd_variance) / 100.0;

    if (lbd_queue_size == 50 && threshold_factor * fast_avg * dynamic_threshold > slow_avg) {
        return true;
    }
    return false;
}
```

Fig. 6. Example of LLM discovered heuristics of restart condition

original restart_condition_update

```
void Solver::restart_condition_update(int &lbd)
{
    if (lbd_queue_size < 50) lbd_queue_size++;       // update fast-slow.
    else fast_lbd_sum -= lbd_queue[lbd_queue_pos];
    fast_lbd_sum += lbd, lbd_queue[lbd_queue_pos++] = lbd;
    if (lbd_queue_pos == 50) lbd_queue_pos = 0;
    slow_lbd_sum += (lbd > 50 ? 50 : lbd);
}
```

discoverd restart_condition_update

```
void Solver::restart_condition_update(int &lbd) {
    static const int LBD_QUEUE_MAX_SIZE = 50;
    static const double FAST_WEIGHT = 0.8;
    static const double SLOW_WEIGHT = 0.2;

    // Update fast LBD sum
    if (lbd_queue_size < LBD_QUEUE_MAX_SIZE) {
        lbd_queue_size++;
    } else {
        fast_lbd_sum -= lbd_queue[lbd_queue_pos];
    }
    fast_lbd_sum += lbd;
    lbd_queue[lbd_queue_pos++] = lbd;
    if (lbd_queue_pos == LBD_QUEUE_MAX_SIZE) {
        lbd_queue_pos = 0;
    }

    // Update slow LBD sum with a more dynamic approach
    double current_lbd_weight = (lbd > LBD_QUEUE_MAX_SIZE) ? LBD_QUEUE_MAX_SIZE : lbd;
    slow_lbd_sum = (slow_lbd_sum * SLOW_WEIGHT) + (current_lbd_weight * FAST_WEIGHT);

    // Adjust the weight dynamically based on the current LBD
    if (lbd < 10) {
        slow_lbd_sum *= 1.05;  // Slightly increase the influence of recent LBDs
    } else if (lbd > 30) {
        slow_lbd_sum *= 0.95;  // Decrease the influence if LBD is high
    }
}
```

Fig. 7. Example of LLM discovered heuristics of restart condition update

original restart_function

```
void Solver::restart() {
    fast_lbd_sum = lbd_queue_size = lbd_queue_pos = 0;
    backtrack(0);
    int phase_rand = rand() % 100;              // probabilistic rephasing
    if ((phase_rand -= 60) < 0)     for (int i = 1; i <= vars; i++) saved[i] = local_best[i];
    else if ((phase_rand -= 5) < 0) for (int i = 1; i <= vars; i++) saved[i] = -local_best[i];
    else if ((phase_rand -= 20) < 0)for (int i = 1; i <= vars; i++) saved[i] = rand() % 2 ? 1 : -1;
}
```

discovered restart_function

```
void Solver::restart() {
    // Dynamic restart based on LBD statistics
    double fast_avg = lbd_queue_size > 0 ? fast_lbd_sum / lbd_queue_size : 0;
    double slow_avg = conflicts > 0 ? slow_lbd_sum / conflicts : 0;

    // Adjust restart aggressiveness based on LBD trends
    double restart_aggressiveness = 0.85;
    if (fast_avg > slow_avg * 1.2) {
        restart_aggressiveness = 0.7; // More aggressive restart when LBD is worsening
    } else if (fast_avg < slow_avg * 0.8) {
        restart_aggressiveness = 0.9; // Less aggressive restart when LBD is improving
    }
    // Partial restart: only backtrack partway based on LBD statistics
    int partial_level = pos_in_trail.size() * restart_aggressiveness;
    backtrack(partial_level);
    // Smart phase selection based on recent activity
    int phase_choice = rand() % 100;
    if (phase_choice < 40) { // 40% chance: use local best phases
        for (int i = 1; i <= vars; i++) saved[i] = local_best[i];
    } else if (phase_choice < 70) { // 30% chance: use inverse of local best
        for (int i = 1; i <= vars; i++) saved[i] = -local_best[i];
    } else if (phase_choice < 85) { // 15% chance: use random phases
        for (int i = 1; i <= vars; i++) saved[i] = rand() % 2 ? 1 : -1;
    } else { // 15% chance: use VSIDS-based phase selection
        for (int i = 1; i <= vars; i++) {
            saved[i] = (activity[i] > 0) ? 1 : -1;
            if (activity[i] == 0) saved[i] = rand() % 2 ? 1 : -1;
        }
    }
    // Reset LBD statistics for new restart period
    fast_lbd_sum = lbd_queue_size = lbd_queue_pos = 0;
}
```

Fig. 8. Example of LLM discovered heuristics of restart function

original rephase_condition

```
bool Solver::rephase_condition() {
    if (rephases >= rephase_limit) return true;
    else return false;
}
```

discovered rephase_condition

```
bool Solver::rephase_condition(){
    // Calculate the ratio of conflicts to restarts to determine solver activity
    double conflict_restart_ratio = static_cast<double>(conflicts) / (restarts + 1);
    // Introduce a dynamic threshold adjustment based on solver's performance
    double dynamic_rephase_threshold = rephase_limit * 1.5;
    // Combine the original condition with the new dynamic component
    if (rephases >= dynamic_rephase_threshold && conflict_restart_ratio > 1.8) return true;
    else return false;
}
```

Fig. 9. Example of LLM discovered heuristics of rephase condition

original reduce_condition

```
bool Solver::reduce_condition() {
    if (reduces >= reduce_limit) return true;
    else return false;
}
```

discovered reduce_condition

```
bool Solver::reduce_condition() {
    // Dynamic reduction condition based on multiple factors
    double memory_pressure = (double)clause_DB.size() / (origin_clauses + 10000);
    double progress_rate = (double)conflicts / (reduce_limit + 1000);
    // Adaptive threshold based on solver state
    if (memory_pressure > 2.0 || progress_rate > 0.8) {
        return reduces >= reduce_limit * 0.7;
    }
    // Standard condition with memory awareness
    return reduces >= reduce_limit && memory_pressure > 1.2;
}
```

Fig. 10. Example of LLM discovered heuristics of reduce condition

original bump_var

```
void Solver::bump_var(int var, double coeff) {
    if ((activity[var] += var_inc * coeff) > 1e100) {           // Update score and prevent float
    overflow
        for (int i = 1; i <= vars; i++) activity[i] *= 1e-100;
        var_inc *= 1e-100;}
    if (vsids.inHeap(var)) vsids.update(var);                 // update heap
}
```

discovered bump_var

```
void Solver::bump_var(int var, double coeff) {
    // Introduce a decay factor to gradually reduce activity over time
    const double decay_factor = 0.95;
    // Update the activity with the coefficient and decay factor
    activity[var] = activity[var] * decay_factor + var_inc * coeff;
    // Check for overflow and scale down if necessary
    if (activity[var] > 1e100) {
        for (int i = 1; i <= vars; i++) {
            activity[i] *= 1e-100;
        }
        var_inc *= 1e-100;
    }
    // Update the heap if the variable is in it
    if (vsids.inHeap(var)) {
        vsids.update(var);
    }
}
```

Fig. 11. Example of LLM discovered heuristics of claBumpActivity

Received 23 September 2025; accepted 15 April 2026