Supplementary Materials

1 Preliminaries

1.1 Background 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 is at least one satisfying assignment. The empty clause $\perp$ is always unsatisfiable and represents a *conflict*. SAT is the problem of deciding whether a given CNF formula is satisfiable.

To solve such the SAT problem, a straightforward approach is to conduct exhaustively search over all possible truth assignments to the variables in V. A complete SAT solver implements this idea by exploring all possible assignments to determine if at least one satisfies F (proving F is satisfiable). If no satisfying assignment exists, the solver conclusively proves F is unsatisfiable. Within this framework, the strategies for branching (for choosing which variable to assign next and what value to assign it) are crucial. Key techniques to enhance the search efficiency include restart strategies, restart strategies, and activity-based heuristics (often implemented by "bumping" variable or clause activity scores).

1.2 CDCL solver

Conflict-Driven Clause Learning (CDCL)[1] is the most common approach and plays a dominant role in modern high-performance SAT solvers. The core innovation is conflict analysis: when a conflict occurs, the solver analyzes it and derives a new clause that explains the inconsistency. These derived clauses, called learned clauses, enable the solver to prune large portions of the search space, which significantly improve the computational efficiency. However, an excessive number of learnt clauses can degrade unit propagation speed and thus consume excessive memory. Consequently, identifying high-quality learne clauses and reducing their quantity are essential for maintaining a solver's performance.

Specifically, CDCL solvers operate on a propagation and learning mechanism, complemented by decision heuristics. The implementation of propagation and learning is a standard practice and overall similar in different solvers. In contrast, the decision policies (for variable selection) and restart policies (for search restarts) which are crucial for performance vary significantly. Due to the need of manual design and the absence of rigorous mathematical proofs for their optimality, these policies are referred to as important heuristics.

There is a long history of research on these heuristics in SAT solvers, among which branching heuristics play a crucial role and continue to impact the performance of CDCL solvers. For example, Variable State Independent Decaying Sum (VSIDS)[2] is a family of branching heuristics that seek to assign a value to the most promising variable in the Make Decision phase. Another important branching heuristic is Learning-Rate Branching (LRB)[3], which frames branching as an optimization problem that picks a variable to maximize a metric called learning rate.

Restart heuristics is also essential for enhancing the performance of CDCL solvers. It allows the solver to abandon the current search path and backtrack to a specific decision level. In this process, the learnt clauses are usually maintained for the next search. Fast restart[4] is a widely used method, and Luby restarts[5] is also heavily used because they represent a prior optimal strategy. However, most implementations are now switched to Glucose-style restarts[6], which are widely used in the SAT Competition. For more details, see the extensive overview given by Armin Biere[7].

Rephase is another technique in CDCL solvers. The primary objective is to reset or adjust the current partial assignment, thereby enabling the solver to explore a diverse search space. PrecoSAT and PicoSAT[8] utilize a Jeroslow-Wang score[9] to adjust the saved phases either on all or only on irreundant clauses in regular intervals following a luby sequence. StrangeNight[10] employs a strategy of flipping values with a certain probability that depends on the depth of the assignment. The motivation is to avoid the heavy-tail phenomenon. These rephasing heuristics have been used and compared in the SAT solver Riss[11].

2 ModSAT: A Modularized SAT solver

2.1 Overview of ModSAT

In this section, details of the proposed modularized SAT solver are provided. ModSAT obeys the basic CDCL framework[1] in Algorithm 1, which usually initiates with an empty set of partial assignments (line 2). The Unit Propagation (UP), also called Boolean Constraint Propagation, assigns values to variable in clauses which has only one variable. UP can always make a clause satisfied, and this operation will repeat until no more UP is possible (line 4). If no conflicts are detected in $\mathscr{X}$

Algorithm 1: CDCL Framework

```
1  Input: A CNF F of SAT instance;
2  Initialization: decision level d ← 0, current assignment of variables 𝒳 ← ∅;
3  while True do
4      𝒳 ← Unit Propagation(F, 𝒳);
5      if Conflicts are detected in 𝒳 then
6          if d == 0 then
7              return UNSAT
8          else
9              𝒞_conflict, d_backtrack ← Analyze Conflict(F, 𝒳) ;
10             C_learned ← Learn Clause(𝒞_conflict, 𝒳) ;
11             F ← F ∧ C_learned;
12             𝒳 ← Backtrack(𝒳, d_backtrack);
13             d ← d_backtrack;
14     else
15         if All variables are assigned with a value then
16             return SAT
17         else
18             d ← d + 1;
19             𝒳 ← Make Decision(𝒳)
```

during the Decision Detection phase, the algorithm will select a variable to assign a value (line 19). This Make Decision step usually contains a few heuristics. When conflicts are detected (line 5), Analyze Conflict will identify the conflicted clauses (line 9), and a newly learnt clause will be derived based on the clause (i.e., the current partial assignment) of conflicts (line 10). Afterwards, 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 satisfied assignment (line 16); otherwise, detecting conflicts at the decision level 0 indicates that the given CNF is unsatisfiable (line 7).

Various heuristics have been proposed in ModSAT to enhance its performance. For instance, *reduce* heuristics identify and remove the learnt clauses by controlling the size of the tracking list. In addition, *bump var heuristics* are usually incorporated in the Analyze Conflict step, which affect the choice of variables in the Make Decision function. While the order of choosing variables can determine the search path of branching, *rephase* heuristics can control the polarity in variales to be selected. Also, *restart* heuristics may abandon the current search path, allowing algorithms to explore possibly easier search regions.

More precisely, in ModSAT, we have defined *seven* functions which are independently implemented: **restart function**, which manages restart heuristics; **restart condition**, which determines when to execute restart; **reduce condition**, which determines when to reduce; **rephase function**, which manages rephase heuristics; **rephase condition**, which determines when to rephase; **bump var activity**, which governs the order of variables selection in Make Decision; and **bump cla activity** that govern the order of clauses being removed during reduce. These functions collectively improve the solver's ability to handle large and complex SAT instances by balancing exploration, exploitation, and resource management. The combination of conflict-driven learning, backtracking, and heuristic enhancements makes ModSAT an efficient and robust SAT solver.

2.2 Three principles behind ModSAT

As already mentioned in the paper, we follow three principles when developing ModSAT to ensure it is LLM-friendly:

- **Maintain functions simple and focus.** The function optimized by LLMs should be simple and explicit, unlike common implementation in complex solvers.
- **Utilize class variables for shared information.** Local variables should be declared as class member variables to give LLMs access to them.
- **Proactively prevent bugs during heuristics discovery.** The bugs written by LLMs should be fixed proactively, so that the solver could compile correctly with the same heuristics, which helps LLMs to generate more diverse correct codes.

The following example demonstrates the application of the principle *Maintain functions simple and focus*, with the original function given in Figure 1. Instead of modifying the whole search function by LLMs, we modularize it into three distinct

Algorithm 2: ModSAT

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

functions by isolating the components that significantly impact the performance and are suitable for LLM to modify, see Figure 2. The decomposition also enables LLMs to focus on refining one kind of heuristic at a time, thereby enhancing the code generation capability.

For the principle *use Class Variables for Shared information*, we move variables that may reside in local scope into class members to ensure that LLMs can access this shared information, see Figure 3.

Figures 4 and 5 provide two example on how to *prevent bugs during heuristics discovery proactively*. The first one is to include some extra packages to prevent LLMs from implementing extra functions, while the second one is to overload common functions to prevent LLMs from incorrectly implementing simple functions because of a misunderstanding of data structures.

3 Automatic Prompt Optimization

To tackle the problem of prohibitive execution time (5000s timeout per instance) and with large performance variance across different datasets, we propose an unsupervised automatic prompt optimization method using Shannon entropy as the evaluation metric. This approach eliminates the dependency on explicit labels while maintaining the adaptability to the evolvement of LLMs, particularly suitable for compute-intensive optimization tasks with ambiguous success criteria.

The basic prompt template follows the instructions in OpenAI framework docs[12], which obeys the following format:

- Define the **Role** of an agent as a solver expert who needs to assess and improve the heuristics function in the SAT solver.

Maintain functions simple: Original function to modify

```
lbool Solver::search(int nof_conflicts){
    // if there is a conflict
    ...... ......
    // if there is no conflict
    if ((lbd_queue_size == 50 && 0.8 * fast_lbd_sum / lbd_queue_size > slow_lbd_sum /
    conflicts) || !withinBudget())
        restart_function();

    // Simplify the set of problem clauses:
    if (decisionLevel() == 0 && !simplify())
        return l_False;

    // Reduce the set of learnt clauses:
        if (learnts.size()-nAssigns() >= max_learnts)
                reduceDB();
    if (rephase_condition())
        rephase_function();

    Lit next = lit_Undef;
    while (decisionLevel() < assumptions.size()){
        // Perform user provided assumption:
        Lit p = assumptions[decisionLevel()];
        if (value(p) == l_True){
            // Dummy decision level:
            newDecisionLevel();
            }else if (value(p) == l_False){
                analyzeFinal(~p, conflict);
                return l_False;
            }else{
                next = p;
                break;
            }
        }

    if (next == lit_Undef){
        // New variable decision:
        decisions++;
        next = pickBranchLit();

        if (next == lit_Undef)
            // Model found:
            return l_True;
    }
    newDecisionLevel();
    uncheckedEnqueue(next);
}
```

Figure 1. Illustration of principle *Maintain functions simple and focus.*

- Clearly state the **Goal**, such as providing optimization suggestions, writing code, or feedback.
- Enhance the agents' 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 codes and results and can be instructed to specify the types of modification directions such as changing parameters, modifying heuristics, or adding new heuristics.

Maintain functions simple: modularized function to modify

```
// original function which has been modularized
lbool Solver::search(int nof_conflicts){
    ...... ......

    if (restart_condition())
        restart_function();
    if (reduce_condition())
        reduceDB();

    if (rephase_condition())
        rephase_function();

    ...... ......
}

// functions to modify
bool Solver::rephase_condition() {
    if (rephases >= rephase_limit)
        return true;
    else
        return false;
}

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

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

Figure 2. Illustration of principle *Maintain functions simple and focus*.

- Key code of SAT solver is appended at the end of each prompt to ensure all agents are in the same context. Note that the key code includes member parameters in cpp class of the solver (LLMs may utilize), along with the main loop function, and all corresponding functions LLMs may need to understand (e.g. such as the functions call the target optimized function).

The previous three components, role, goal and tips can be automatically optimized by LLMs. In each iteration, we randomly select one component to optimize by LLMs, compute the correctness and diversity (entropy) of the generated codes, and update the component in prompt with better performance in both correctness and diversity. This loop will iterate until the stopping criteria is met, see Algorithm 3 for details. An empirical evaluation of the original and optimized prompts is presented in Figure 6. It is clear that optimized prompt achieves better performance than the original one.

4 Presearch strategy

To overcome this combinatorial explosion when searching over all candidate functions, we propose a novel two-phase optimization strategy:

Add class member variables

```
// LBD heuristics
int lbd_queue[500],   // circled queue saved the recent 500 LBDs.
    lbd_queue_size,   // The number of LBDs in this queue
    lbd_queue_pos;
double fast_lbd_sum, slow_lbd_sum;

// rephase heuristics
int rephases, rephase_limit, rephase_count, threshold;
double last_rephase_progress;

// restart heuristics
int curr_restarts;
double last_restart_progress;
```

Figure 3. Illustration of principle *Utilize class variables for shared information*

Prevent bugs: add more packages

```
#include <math.h>
#include <unordered_set>
#include <algorithm>
using namespace std;
```

Figure 4. Illustration of principle *Proactively prevent bugs during heuristics discovery*

- Presearh function candidate: Conduct small-scale preliminary testing to identify and eliminate functions that consistently degrade the performance.
- Refined evolutionary search: Execute a focused $(1+\lambda)$ Evolutionary Algorithm (EA) search only on the high-impact candidate functions identified in the last phase.

A detailed description of this strategy can be found in Algorithm 4. For the phase of presearh function candidate, we construct a compact and representative evaluation subset comprising the 50% problem instances from the original dataset. This subset can capture the essential solver behavior while minimizing the evaluation cost. Each candidate function is evaluated separately on the sub-dataset for its impact on the solver's PAR-2 score (where lower values indicate better performance), which measures the standalone effect of adding the function to a baseline solver. Then functions that degrade the PAR-2 score are identified and pruned. For each dataset, the pruned functions will be not be further considered for evolutionary search. This phase typically retains a small set of high-impact functions (e.g., 4 functions) that consistently show a positive or neutral effect on PAR-2 in the preliminary tests. For the evolutionary optimization phase, we use the significantly refined function set (e.g., about 4 functions) to execute a standard $(1+\lambda)$ Evolutionary Algorithm on the full target dataset to find optimal combinations.

It is also helpful to investigate the contribution of each function to the final results. To this end, we have calculated the number of times that a function contributes to the final performance improvement in each experiment, see Figure 8, where each subfigure shows the results for one dataset. It can be observed that almost all functions we select could contribute substantially to the final performance,

5 Heuristics discovery

The details of heuristics discovery in AutoModSAT in presented in Figure 9, which contains three agents, LLMs coder, LLMs evaluator, LLMs repairer. Since the LLM coder agent is already optimized in automatic prompt optimization, here we give a brief discussion of the LLMs evaluator and LLMs repairer.

In our experiments, we find that sometimes LLMs coder would generate synonymous code compared with original one, even though we have optimized the prompt. This will lead to redundant iterations in the evolutionary search. Thus, we

Prevent Bugs: overloading functions

```cpp
#include <type_traits>

template <typename T1, typename T2>
typename std::common_type<T1, T2>::type max(T1 a, T2 b) {
    static_assert(std::is_integral<T1>::value && std::is_integral<T2>::value,
                  "max: Both types must be integers (int or long int)");
    return (a < b) ? b : a;
}

template <typename T1, typename T2>
typename std::common_type<T1, T2>::type min(T1 a, T2 b) {
    static_assert(std::is_integral<T1>::value && std::is_integral<T2>::value,
                  "min: Both types must be integers (int or long int)");
    return (a < b) ? a : b;
}

template <typename T1, typename T2>
typename std::common_type<T1, T2>::type max(T1 a, T2 b) {
    static_assert(std::is_floating_point<T1>::value && std::is_floating_point<T2>::value,
                  "max: Both types must be floating-point (float or double)");
    return (a < b) ? b : a;
}

template <typename T1, typename T2>
typename std::common_type<T1, T2>::type min(T1 a, T2 b) {
    static_assert(std::is_floating_point<T1>::value && std::is_floating_point<T2>::value,
                  "min: Both types must be floating-point (float or double)");
    return (a < b) ? a : b;
}
```

Figure 5. Illustration of principle *Proactively prevent bugs during heuristics discovery.*

develop an LLMs evaluator agent, which identifies whether the generated code is synonymous. If the code is synonymous, LLMs coder needs to regenerate a code; otherwise, the code will be sent to compile.

In addition, LLMs sometimes make mistakes, such as missing parentheses or utilizing incorrect data types. To address this issue, an LLM repairer agent is introduced. LLM repairer analyzes the LLM-generated code and the corresponding errors, and then seeks to fix the bug. This agent can successfully resolves some common errors. While there are some errors remaining uncorrectable, LLM Coder will re-generate the code.

6 Experimental details

6.1 Dataset description

Here we give more details about the datasets, including 7 datasets selected from SAT Competition 2023 and 2024, 3 generated ones by Picat, and another one from an industrial EDA scenario.

- **Argumentation problem** involves finding acceptable sets of arguments in a directed graph where attacks between arguments are represented by edges.
- **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.

Algorithm 3: Automatic Prompt Optimization

Input: initial prompt template P, solver codebase S, max_iterations i, prompt optimized part $R = \{Role, Goal, Tips\}$
Output: optimized prompt template P^*

$i \leftarrow 10$ // prompt optimization iterations
$j \leftarrow 20$ // number of code generation in each iteration
while $i \geq 0$ **do**
  select prompt part r from R uniformly at random
  $P' \leftarrow$ refine_prompt(r, P) // LLMs refine the prompt template
  while $j \geq 0$ **do**
    generated code $c \leftarrow$ call_llm$(current_prompt)$
    compilation error $e \leftarrow$ compile(c, S)
    if $\neg e$ **then**
      $C \leftarrow C \cup c$
    else
      execute_code$(corrected_code)$
    end
    $j \leftarrow j - 1$
  end
  diversity $d_i \leftarrow$ compute_code_diversity(C)
  success rate $s_i \leftarrow$ compute_code_success(C)
  if $d_i > d$ *and* $s_i > threshold$ **then**
    $d \leftarrow d_i$
    $P \leftarrow$ update_prompt(S, P')
  else
  end
  $i \leftarrow i - 1$
end

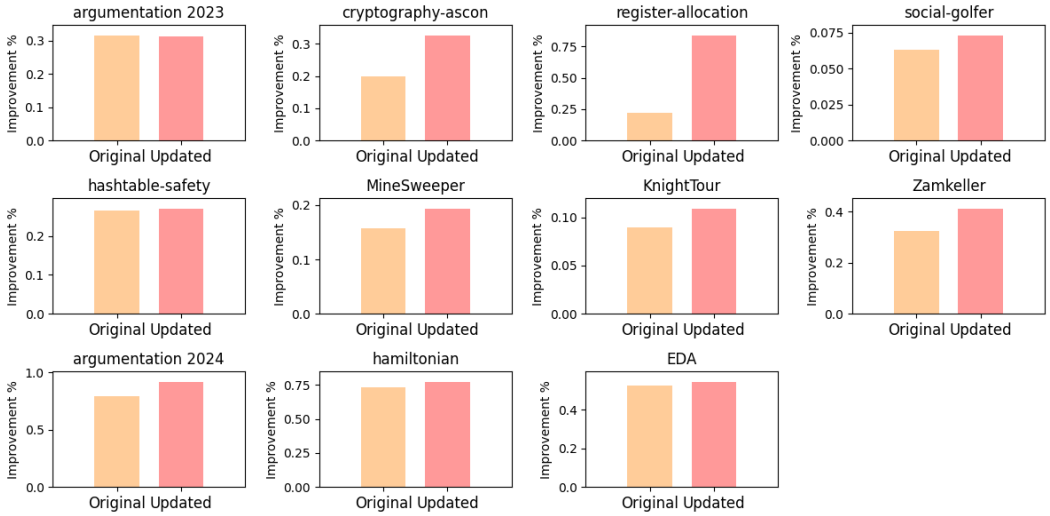

Figure 6. Comparison over Different Prompts. This figure shows the performance of AutoModSAT using different prompts. The vertical axis shows the PAR-2 improvement ratio compared to the original solver, while the two coordinates in the horizontal axis correspond to the original prompt (Original) and optimized prompt (Updated), respectively.

- **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.
- **Cryptography-Ascon problem** is a lightweight cryptographic algorithm challenge focused on implementing and ver-

Original Prompt

(Role) You are a SAT solver researcher trying to rewrite the {{ func_name }} function(s).

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

(Tips) Tips:
1) Your rewrited function code must start with "'// start function name"' and end with "'// end function name"'
2) Your rewrited function(s) code must be different from original code, not just rewrite code synonymous!
3) You are not allowed to create your own new function(s) in the rewrited 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 rewrited function(s) code can be executed correctly.

<key code> of SAT solver is:
{{ replace_key_code }}

Updated Prompt

(Role) You are a SAT solver researcher trying to improve the {{ func_name }} function.

(Goal) Objective:
Your goal is to improve the SAT solver by rewriting the {{ func_name }} function.

Instructions:
1. Carefully read and comprehend the <key code> of the SAT solver provided below.
2. Analyze potential improvements and devise a strategy for optimizing the heuristics of function.
3. Deliver your improved function(s) with the following format:
- Begin with: '// start function name'
- End with: '// end function name'

(Tips) Tips:
1. Ensure that your rewritten function(s) are substantially different from the original, beyond mere synonym replacements.
2. You may utilize existing global variables from the <key code>, but refrain from introducing new global variables.
3. Verify that the rewritten function(s) execute correctly.

Take a deep breath and think it step by step.

<key code> of SAT solver is:
""" {{ replace_key_code }} """ ...

Figure 7. Comparison of original prompt and optimized prompt

ifying the Ascon cipher, a NIST-standardized authenticated encryption scheme, for resource-constrained IoT devices, balancing security against differential attacks with minimal computational overhead.

- **Hamiltonian problem** is a graph theory problem that involves determining whether a given graph contains a Hamiltonian cycle, a closed loop visiting each vertex exactly once, which is NP-complete and often applied to route optimization or circuit design verification.
- **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

Algorithm 4: PreSearch Strategy in AutoModSAT

Input: Datasets P, modularized SAT solver with seven functions $\{h_1, \ldots, h_7\}$, prompt template, baseline functions $\{b_1, \ldots, b_7\}$
Phase 1: PreSearch Function Candidate $P_{\text{compact}} \leftarrow 50\%$ representative instances from P
$R \leftarrow \emptyset$; // Retained function set
for *each function* $h_i \in \{h_1, \ldots, h_7\}$ **do**
  $A_{\text{test}} \leftarrow$ build solver replacing h_i with baseline b_i
  $f_i, s_i \leftarrow$ evaluate(A_{test}, P_{compact}) ; // Get PAR-2 metric
$F \leftarrow \text{sort}(f_1, ..., f_7)$; // Sort PAR-2 metric in different functions
Get top 4 function index R from F
Phase 2: Evolutionary Algorithm Optimization $A \leftarrow$ solver with functions: $(\forall i \in R : h_i) \cup (\forall i \notin R : b_i)$
$f^* \leftarrow$ evaluate(A, P) ; // Full dataset evaluation
evalBudget $\leftarrow 50$; // Maximum evaluations
while *evalBudget* > 0 **do**
  $M \leftarrow \emptyset$
  $\ell \sim \text{Bin}(|R|, \frac{1}{|R|})$; // Sample modification count
  chosen ℓ distinct values $M \leftarrow \{m_0, \ldots, m_\ell\}$ from R uniformly at random;
  Generate new functions $\{h'_m\}_{m \in M}$ via LLM using A and M
  $A' \leftarrow$ update A with $\{h'_m\}_{m \in M}$
  $f(A') \leftarrow$ evaluate(A', P)
  if $f(A') \leq f^*$ **then**
    $A \leftarrow A'$
    $f^* \leftarrow f(A')$
  evalBudget $\leftarrow$ evalBudget $- 1$

$1, 1, 2, 2, ..., n, n$ 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 $(1 < k < n)$.
- **EDA problem** involves formally proving whether two design specifications are functionally equivalent, which is one of the most essential techniques in Electronic Design Automation and digital IC design. It has a wide range of applications, such as functional equivalent logic removal, sequential equivalence checking, circuit-based method for symmetries detection, engineering change orders, among others.

For the generated instances using Picat[13], we adopt the settings in Chapters 2 and 3 of the book by NengFa Zhou[13] and conduct grid sampling within a parameter space. Specifically, the parameter space $\Theta = \{\theta_1, \theta_2, \ldots\}, \theta_i^L \leq \theta_i \leq \theta_i^U$ is defined to ensure that three baseline solvers including EasySAT, MiniSat and Kissat, can obtain a solution within proper cputime range, i.e., [1s, 5000s]. The we apply a space Θ' to generate the dataset by enlarging the upper bound of Θ, e.g., $\theta_i'^U = \theta_i'^U * 1.2$, such that we can test whether AutoModSAT can solve the instances where the baseline solvers cannot.

- **MineSweeper problem**
  Parameter Θ: $\{m, n, k, p\}$
  Parameter Space: $[500, 1, 600] \times [400, 3200] \times [72, 689, 1, 572, 118] \times [0.32, 0.38]$
  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).
- **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.)

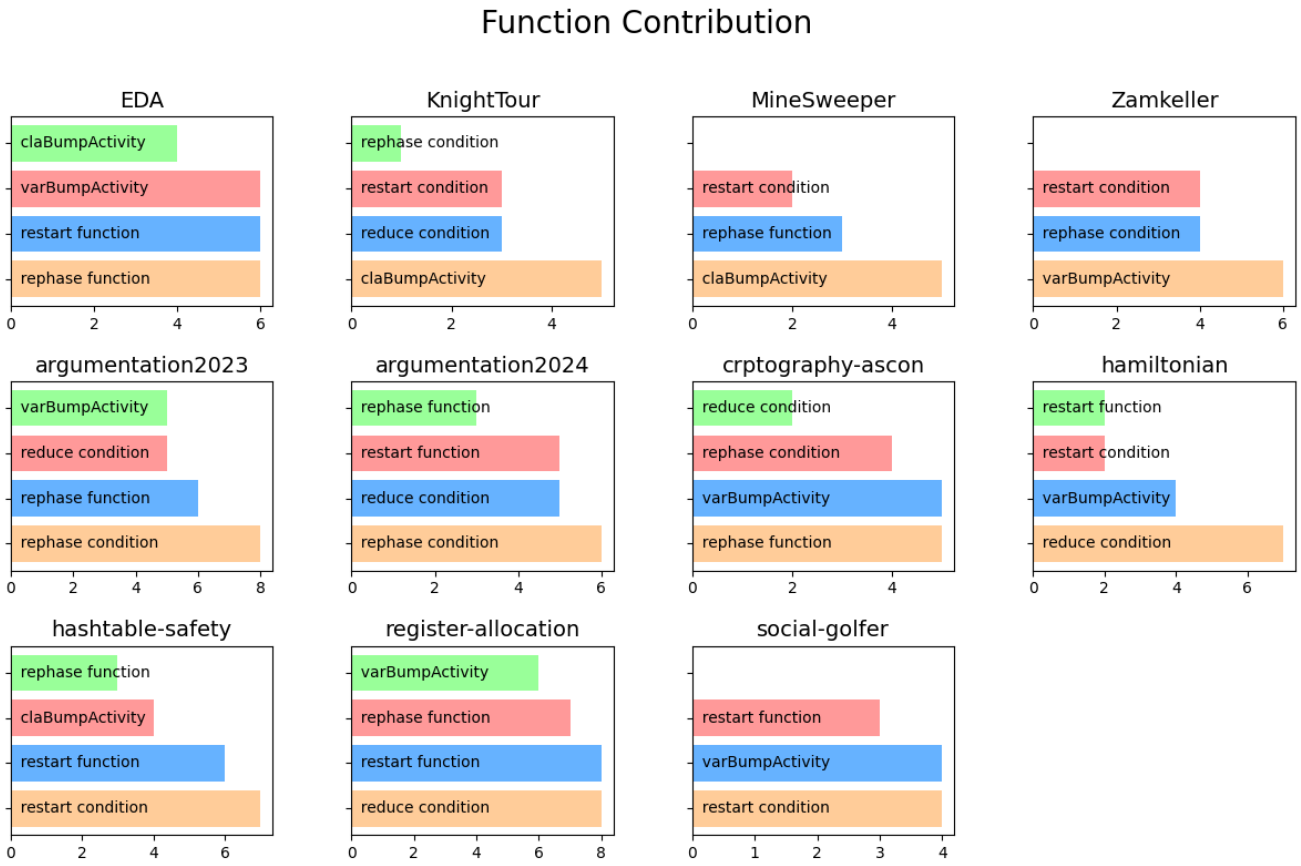

Figure 8. Function Contribution: where the vertical axis lists different function candidates and the horizontal axis denotes the frequency of contributions.

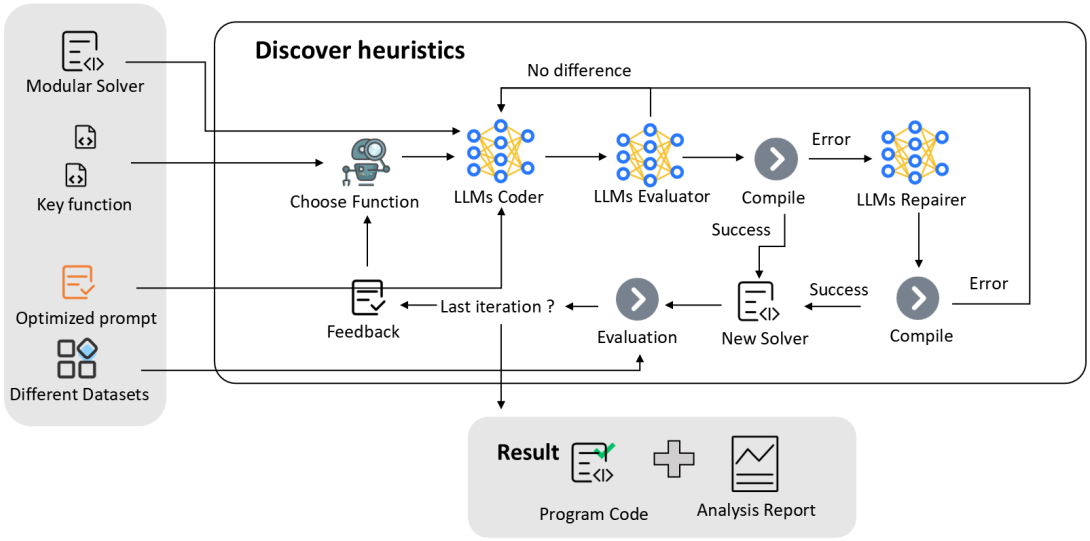

Figure 9. Details about heuristics discovery

- **Zamkeller problem**:
  Parameter Θ: $\{k, n\}$
  Parameter Space: $[3, 34] \times [25, 100]$
  Notes: k represents the total sequence length, and n represents the subsequence length. For all subsequences of length k, the goal is to change them into the minimum number of distinct sequences.

The detailed configuration of the training dataset and the function candidates in heuristics discovery are presented in Table 1 .

6.2 Evaluation Metric

We consider two specific metrics for evaluation of a SAT solver: (1) the number of SAT instances solved within the given timeout bound, and (2) the Penalized Average Runtime with a factor of 2 score (PAR-2). Both metrics are commonly used in the SAT Competitions.

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

$$\text{PAR-2} = \frac{1}{n}\sum_{i=1}^{n}\tau_i \quad \text{where:} \quad \tau_i = \begin{cases} t_i, & \text{if } t_i \leq \mathscr{T} \\ 2\mathscr{T}, & \text{if } t_i > \mathscr{T} \text{ or the solver fails to return a result,} \end{cases}$$

where $\mathscr{T}$ is the predefined timeout bound. For example, consider a benchmark dataset with three instances and a timeout bound $\mathscr{T} = 100$ seconds. The runtimes (in seconds) for the three instances are: $t_1 = 80$ for instance 1, $t_2 = 120$ for instance 2, the solver fails to return a result for instance 3. Then

- since $t_1 \leq \mathscr{B}$, we have $\tau_1 = 80$;
- since $t_2 > \mathscr{B}$, we have $\tau_2 = 200$ (penalized);
- since the solver fails for instance 3, we have $\tau_3 = 200$ (penalized).

Therefore, The PAR-2 score is given by

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

Table 1. Configuration of training set, where the indices 1 to 7 represent the following function candidates in order: rephase_condition, rephase_function, reduce_condition, restart_condition, restart_function, varBumpActivity, claBumpActivity.

Dataset	Training Timeout	Function candidate
cryptography-ascon	800	1,2,3,6
register-allocation	5000	2,3,5,6
social-golfer	2000	1,4,5,6
hashtable-safety	500	2,4,5,7
argumentation 2023	2000	1,2,3,6
argumentation 2024	2000	1,2,3,5
hamiltonian	800	3,4,5,6
MineSweeper	500	2,4,3,7
KnightTour	2000	1,3,4,7
Zamkeller	2000	1,3,4,6
EDA	800	2,5,6,7

6.3 Search Space for Parameter Tuning

In this paper, we adopt SMAC3 to optimize the baseline SAT solver parameters across different datasets. The search space for each solver, including the parameter name, type, description and range, are presented in Tables 2, 3, and 4.

Table 2. ModSAT 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	$\{\text{true},\text{false}\}$
`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 3. 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	$\{6,7,8,9,10,20,50\}$

7 More Examples 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 for the changes.

Figure 10 provides an example for the updated claBumpActivity function, which introduces two key enhancements in contrast to the original implementation. First, during activity rescaling, it enforces a minimum activity threshold ($min\_activity = 1e-20$) to prevent clauses from becoming numerically insignificant after scaling. This preserves the relevance of historically important clauses and avoids premature elimination from the learning process. Second, it incorporates dynamic decay adjustment based on recent conflict rates: when conflicts exceed 1000 and the LBD queue surpasses 50 entries, *cla_inc* is scaled down proportionally to the conflict density (with a floor of 0.8). This adaptively moderates activity growth during high-conflict phases, prioritizing recent impactful clauses while maintaining stability. Together, these refinements yield a more balanced clause management strategypreventing underutilization of valuable learned clauses while dynamically optimizing activity decay for solver efficiency.

Figure 11 provides an example of the updated varBumpActivity function, which introduces three key improvements over the original. First, it scales the increment by (1.0 + 0.1 * decisionLevel()), prioritizing variables involved in recent decisions to accelerate conflict-driven learning. Second, the rescaling mechanism uses a larger threshold (1e100) and finer scale factor (1e-100), while preserving variable relevance by enforcing a minimum activity floor (1e-100) to maintain relative ordering and prevent premature underflow. Third, it optimizes heap management through conditional updates: dynamically adjusting the variable's heap position only when its activity exceeds the current maximum, or inserting undefined decision variables lazily.

Table 4. CaDiCaL configuration parameters

Parameter	Type	Description	Search Space
`chrono`	int	Chronological backtracking mode (0: none, 1: limited, 2: always)	$\{0,1,2\}$
`elim`	bool	Enables variable elimination during simplification	$\{0,1\}$
`forcephase`	bool	Forces phase saving for decision variables	$\{0,1\}$
`minimize`	bool	Enables clause minimization during conflict analysis	$\{0,1\}$
`phase`	bool	Initial decision phase assignment (0: negative, 1: positive)	$\{0,1\}$
`probe`	bool	Enables probing (failed literal detection)	$\{0,1\}$
`reduceint`	int	Conflict interval for clause database reduction	$\{10^2,10^3,10^4,10^5\}$
`rephaseint`	int	Conflict interval for resetting variable phases	$\{10^1,10^2,10^3,10^4,10^5\}$
`restartint`	int	Base restart interval (conflicts between restarts)	$\{2,10^2,10^3,10^4\}$
`restartmargin`	int	Restart margin percentage (Luby sequence scaling)	$\{0,5,10,15,20,25\}$
`stabilize`	bool	Stabilizes search by limiting activity updates	$\{0,1\}$
`target`	int	Search target (0: SAT, 1: UNSAT, 2: balanced)	$\{0,1,2\}$

These enhancements collectively improve search guidance, reduce floating-point stability issues, and minimize unnecessary data structure operations.

Figure 12 provides an example of the updated restart_condition function significantly improves upon the original by replacing its static threshold approach with a dynamic, performance-driven restart strategy that adapts to real-time solver behavior. Instead of relying on fixed queue sizes and hardcoded multipliers, the new version intelligently calculates restart thresholds using multiple runtime metrics: it combines average LBD (measuring clause quality) with conflict rates (tracking solver progress) to dynamically adjust restart timing based on problem difficulty. Crucially, it introduces a progress-sensitive mechanism that aggressively lowers thresholds when stagnation is detected (progressEstimate changes < 0.01), enabling proactive recovery from plateausa capability absent in the original. This multi-factor approach yields more precise restart decisions, reduces wasteful recomputations, and enhances solver adaptability across diverse SAT instances while maintaining robustness during initialization through default thresholds.

Figure 13 provide an example of the updated restart_function, which introduces significant improvements over the original implementation by incorporating adaptive restart strategies based on real-time solver performance metrics. Unlike the original version, which always resets to decision level 0 (a full restart), the enhanced function dynamically calculates two exponential moving averages of conflict difficulty (fast_avg and slow_avg) using Literal Block Distance (LBD) scores. By analyzing the ratio between these averages, it intelligently selects one of three restart depths: full restart (level 0) for deteriorating conflict quality, partial restart (mid-level) for moderately harder conflicts, or minimal restart (current level -1) for stable conditions. This adaptability preserves useful learned clauses during partial/minimal restarts, reducing redundant recomputation. Additionally, periodic clause database reduction (every 16 restarts) curbs memory growth, while rebuilding the variable order heap ensures branching decisions reflect updated activity scores. Collectively, these optimizations balance exploration and exploitation, enhancing solver efficiency through context-aware restarts and resource management.

Figure 14 provides and examples of the updated rephase_condition function, which introduces an adaptive rephasing mechanism that significantly enhances the original static threshold approach. Unlike the prior version which solely relied on a fixed rephase limit, the new implementation dynamically adjusts rephasing intervals based on real-time search progress and conflict density. By calculating normalized progress through trail size changes and setting variable-driven thresholds (e.g., 2% of total variables), it detects stagnation when progress falls below expectations and responds by reducing subsequent rephase intervals exponentially. Conversely, substantial progress triggers gradual interval expansion. This self-tuning capability optimizes computational efficiency: it minimizes unnecessary rephasing during productive search phases while aggressively countering stagnation, thereby improving solution convergence without compromising robustness.

Figure 16 provides an example of the updated rephase_function, which introduces several key improvements over the original implementation, enhancing adaptability and search efficiency. First, it implements dynamic rephase limit adjustment by scaling rephase_limit based on progress measured through conflict resolution (conflictR). If progress occurs, the limit increases by 50% to exploit productive phases more aggressively; otherwise, it decays by 10% (with a lower bound of 512) to conserve resources during stagnation. This replaces the originals static increment (+= 8192) and fixed decay (threshold *= 0.9),

enabling context-sensitive resource allocation. Second, the refined phase selection strategy uses weighted probabilities with four distinct policies: local-best phases (40%), global phase inversion (30%), randomized phases for low-activity variables (20%), and user-specified phases (10%). This replaces the originals rigid three-policy cascade, adding targeted randomization for less-active variableswhich helps escape local optimaand reintroducing user phases for domain-specific guidance. Finally, adaptive threshold reset (threshold = trail.size() * 0.8) dynamically scales with the solvers state, replacing the fixed decay, while verbosity-controlled logging aids debugging. These changes collectively improve the solvers ability to balance exploration versus exploitation, mitigate stagnation, and leverage problem-specific knowledge.

Figure 17 provides an example of the updated reduce_condition function, which significantly enhances the original version through four key improvements that collectively the optimize memory management and solver adaptability. First, it retains the core check for absolute learnt clause limits (learnts.size() >= max_learnts), ensuring baseline constraint adherence. Second, it introduces memory pressure awareness by triggering reduction when wasted clause memory exceeds 80% of the garbage collection threshold (ca.wasted() > ca.size() * garbage_frac * 0.8). This proactively mitigates memory bloat and improves cache efficiency. Third, a learnt-to-original clause ratio check (learnts.size() > 2 * nClauses()) prevents learnt clauses from disproportionately dominating the formula, maintaining balanced reasoning. Finally, a conflict-driven heuristic (conflictR > 1000 && learnts.size() > max_learnts * 0.8) adapts to high-conflict phases by initiating earlier reduction, thus accelerating recovery from solver stagnation. These layered criteria synergistically boost robustness: they minimize redundant computation through memory-sensitive garbage collection, preserve clause quality via ratio controls, and dynamically respond to runtime behaviorultimately yielding faster, more memory-efficient SAT solving.

References

1. Marques-Silva, J. P. & Sakallah, K. A. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Comput.* **48**, 506–521 (1999).
2. Biere, A., Heule, M., van Maaren, H. & Walsh, T. Conflict-driven clause learning sat solvers. *Handb. Satisf. Front. Artif. Intell. Appl.* **4**, 131–153 (2009).
3. Liang, J. H., Ganesh, V., Poupart, P. & Czarnecki, K. Learning rate based branching heuristic for SAT solvers. In *Theory and Applications of Satisfiability Testing–SAT 2016: 19th International Conference*, 123–140 (2016).
4. Ramos, A., Van Der Tak, P. & Heule, M. J. Between restarts and backjumps. In *Theory and Applications of Satisfiability Testing-SAT 2011: 14th International Conference, SAT*, 216–229 (2011).
5. Luby, M., Sinclair, A. & Zuckerman, D. Optimal speedup of las vegas algorithms. *Inf. Process. Lett.* **47**, 173–180 (1993).
6. Audemard, G. & Simon, L. Glucose 2.1: Aggressive, but reactive, clause database management, dynamic restarts. In *Pragmatics of SAT 2012* (2012).
7. Biere, A. & Fröhlich, A. Evaluating cdcl restart schemes. In *Proceedings of Pragmatics of SAT*, 1–17 (2015).
8. Biere, A. Lingeling, plingeling, picosat and precosat at sat race 2010. *FMVReport Ser. Tech. Rep.* **10** (2010).
9. Jeroslow, R. G. & Wang, J. Solving propositional satisfiability problems. *Annals mathematics Artif. Intell.* **1**, 167–187 (1990).
10. Soos, M. Strangenight. In *Proceedings of SAT Competition 2013: Solver, Benchmark and Proof Checker Descriptions*, 89–90 (Department of Computer Science, University of Helsinki, 2013).
11. Balint, A., Belov, A., Järvisalo, M. & Sinz, C. Overview and analysis of the sat challenge 2012 solver competition. *Artif. Intell.* **223**, 120–155 (2015).
12. OpenAI. OpenAI API Documentation. https://platform.openai.com/docs (2023).
13. Zhou, N.-F., Kjellerstrand, H. & Fruhman, J. *Constraint solving and planning with Picat*, vol. 11 (Springer, 2015).

original claBumpActivity

```
inline void Solver::claBumpActivity (Clause& c) {
        if ( (c.activity() += cla_inc) > 1e20 ) {
            // Rescale:
            for (int i = 0; i < learnts.size(); i++)
                ca[learnts[i]].activity() *= 1e-20;
            cla_inc *= 1e-20; }
        }
```

discovered claBumpActivity

```
inline void Solver::claBumpActivity (Clause& c) {
    // Adaptive scaling based on clause length and conflict participation
    if ((c.activity() += cla_inc) > 1e20) {
        // Maintain minimum activity while scaling to preserve relevance
        double scale_factor = 1e-20;
        double min_activity = 1e-20;
        for (int i = 0; i < learnts.size(); i++) {
            ca[learnts[i]].activity() *= scale_factor;
            if (ca[learnts[i]].activity() < min_activity)
                ca[learnts[i]].activity() = min_activity;
        }
        cla_inc *= scale_factor;

        // Stabilize scaling by maintaining lower bound on clause activity
        if (cla_inc < min_activity)
            cla_inc = min_activity;
    }
    // Dynamic decay adjustment based on recent conflict rate
    if (conflicts > 1000 && lbd_queue_size > 50) {
        double conflict_scale = 1.0 - 0.01 * (lbd_queue_size / 50.0);
        cla_inc *= conflict_scale > 0.8 ? conflict_scale : 0.8;
    }
}
```

Figure 10. Example of LLM discovered heuristics of claBumpActivity

original varBumpActivity

```
inline void Solver::varBumpActivity(Var v, double inc) {
    activity[v] += inc;
    // Rescale condition to prevent floating-point overflow earlier
    if (activity[v] > 1e50) {
        for (int i = 0; i < nVars(); i++)
            activity[i] *= 1e-50;
        var_inc *= 1e-50;
        }
    }
```

discoverd varBumpActivity

```
inline void Solver::varBumpActivity(Var v, double inc) {
    // Scale the increment based on decision level to prioritize recent variables
    double scaled_inc = inc * (1.0 + 0.1 * decisionLevel());
    activity[v] += scaled_inc;
    // More gradual rescaling that preserves relative ordering better
    if (activity[v] > 1e100) {
        double scale_factor = 1e-100;
        for (int i = 0; i < nVars(); i++) {
            activity[i] *= scale_factor;
            // Prevent variables from becoming completely inactive
            if (activity[i] < 1e-100) activity[i] = 1e-100;
        }
        var_inc *= scale_factor;
    }
    // Optimized heap update with early exit for common case
    if (order_heap.inHeap(v)) {
        if (activity[v] > activity[order_heap[0]]) {
            order_heap.decrease(v);
        }
    } else if (decision[v] && value(v) == l_Undef) {
        order_heap.insert(v);
    }
}
```

Figure 11. 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) || !withinBudget())
        return true;
    else
        return false;

}
```

discoverd restart_condition

```
bool Solver::restart_condition() {
    // Base restart condition on conflict count
    if (conflicts <= 0) return false;

    // Dynamic restart threshold based on recent solver performance
    double restart_threshold;

    if (lbd_queue_size > 0) {
        // Use both average LBD and conflict rate for adaptive restart
        double avg_lbd = fast_lbd_sum / lbd_queue_size;
        double conflict_rate = (double)conflictR / (double)conflicts;

        // Adjust threshold based on problem difficulty (higher LBD = harder problem)
        restart_threshold = restart_first * (0.8 + 0.4 * avg_lbd) * (1.0 + 0.5 *
    conflict_rate);

        // More aggressive restart if we're making little progress
        if (progressEstimate() - last_rephase_progress < 0.01) {
            restart_threshold *= 0.7;
        }
    } else {
        // Default threshold for initial phase
        restart_threshold = restart_first;
    }

    // Check if we've reached the threshold
    if (conflictR >= restart_threshold) {
        conflictR = 0;
        return true;
    }
    return false;
}
```

Figure 12. Example of LLM discovered heuristics of restart condition

original restart_function

```
void Solver::restart_function(){
    fast_lbd_sum = lbd_queue_size = lbd_queue_pos = 0;
    int level = 0;
    cancelUntil(level);
}
```

discovered restart_function

```
void Solver::restart_function() {

    // Update restart statistics
    static double fast_avg = 0, slow_avg = 0;
    static int restart_count = 0;

    // Calculate moving averages of LBD scores
    if (lbd_queue_size > 0) {
        double curr_fast = fast_lbd_sum / lbd_queue_size;
        fast_avg = 0.9 * fast_avg + 0.1 * curr_fast;
        slow_avg = 0.99 * slow_avg + 0.01 * curr_fast;
    }

    // Determine restart level based on recent performance
    int restart_level = 0;
    if (fast_avg > 0 && slow_avg > 0) {
        double ratio = fast_avg / slow_avg;
        if (ratio > 1.2) {
            // Recent conflicts are harder - do a full restart
            restart_level = 0;
        } else if (ratio > 1.0) {
            // Partial restart - keep some learned clauses
            restart_level = std::max(0, decisionLevel() / 2);
        } else {
            // Recent conflicts are easier - minimal restart
            restart_level = std::max(0, decisionLevel() - 1);
        }
    }

    // Reset LBD statistics
    fast_lbd_sum = lbd_queue_size = lbd_queue_pos = 0;

    // Perform the restart
    cancelUntil(restart_level);

    // Occasionally reduce the clause database after restarts
    if (restart_count++ \% 16 == 15) {
        reduceDB();
    }

    // Rebuild the order heap to reflect any activity changes
    rebuildOrderHeap();
}
```

Figure 13. 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() {
    // Adaptive rephasing based on progress and conflict density
    const int base_rephase_limit = 1024;
    const double progress_factor = 0.02;  // 2% of total variables

    if (conflictR < rephase_limit)
        return false;

    // Calculate normalized progress since last rephase
    int progress = trail.size() - last_rephase_progress;
    int progress_threshold = std::max(50, (int)(nVars() * progress_factor));

    // Dynamic limit adjustment based on search progress
    if (progress < progress_threshold) {
        // Stagnation detected - reduce next rephase interval
        rephase_limit = std::max(base_rephase_limit, rephase_limit * 2/3);
    } else {
        // Good progress - expand next rephase interval
        rephase_limit = std::min(base_rephase_limit * 16, rephase_limit * 3/2);
    }

    // Reset conflict counter and update progress tracker in rephase_function
    return true;
}
```

Figure 14. Example of LLM discovered heuristics of rephase condition

original rephase_function

```
void Solver::rephase_function() {
    int var_nums = nVars();
    conflictR = 0, rephases = 0, threshold *= 0.9, rephase_limit += 8192;Add commentMore
    actions
    int phase_rand = rand() \% 100;
    if ((phase_rand -= 40) < 0){
        for (int i = 0; i < var_nums; i++){
            polarity[i] = local_best[i];
        }
    }
    else if ((phase_rand -= 25) < 0){
        for (int i = 0; i < var_nums; i++){
            polarity[i] = 1-local_best[i];
        }
    }
    else if ((phase_rand -= 15) < 0){
        for (int i = 0; i < var_nums; i++){
            polarity[i] = !polarity[i];
        }
    }
    else {
        for (int i = 0; i < var_nums; i++){
            polarity[i] = saved[i];
        }
    }
}
```

Figure 15. Example of LLM discovered heuristics of rephase function

discovered rephase_function

```
void Solver::rephase_function() {
    // Adaptive rephase limit adjustment
    if (rephases > 0 && conflictR > last_rephase_progress) {
        // Progress was made - be more aggressive with rephasing
        rephase_limit = rephase_limit * 1.5;
    } else {
        // No progress - be more conservative
        rephase_limit = rephase_limit * 0.9;
        if (rephase_limit < 512) rephase_limit = 512;
    }
    last_rephase_progress = conflictR;
    rephase_count++;

    // Dynamic phase selection strategy
    double rand_val = drand(random_seed);
    if (rand_val < 0.4) {
        // 40\% chance: Use local best phases
        for (int v = 0; v < nVars(); v++) {
            polarity[v] = local_best[v];
        }
        if (verbosity >= 1) printf("| Rephase: Local best phases |\n");
    } else if (rand_val < 0.7) {
        // 30\% chance: Invert all phases
        for (int v = 0; v < nVars(); v++) {
            polarity[v] = !polarity[v];
        }
        if (verbosity >= 1) printf("| Rephase: Inverted all phases |\n");
    } else if (rand_val < 0.9) {
        // 20\% chance: Random phases for variables with low activity
        double activity_threshold = 0.2 * var_inc;
        for (int v = 0; v < nVars(); v++) {
            if (activity[v] < activity_threshold) {
                polarity[v] = drand(random_seed) < 0.5;
            }
        }
        if (verbosity >= 1) printf("| Rephase: Randomized low-activity phases |\n");
    } else {
        // 10\% chance: Original user phases
        for (int v = 0; v < nVars(); v++) {
            if (user_pol[v] != l_Undef) {
                polarity[v] = (user_pol[v] == l_True);
            }
        }Add commentMore actions
        if (verbosity >= 1) printf("| Rephase: Reset to user phases |\n");
    }

    // Reset the threshold for next local best phase tracking
    threshold = trail.size() * 0.8;
    cancelUntil(0);
}
```

Figure 16. Example of LLM discovered heuristics of rephase function

original reduce_condition

```
bool Solver::reduce_condition()
{
    if (learnts.size() >= max_learnts)  return true;
    else return false;
}
```

discovered reduce_condition

```
bool Solver::reduce_condition()
{
    // Check if we've reached the absolute limit of learnt clausesAdd commentMore actions
    if (learnts.size() >= max_learnts) return true;

    // Consider memory pressure and garbage collection needs
    if (ca.wasted() > ca.size() * garbage_frac * 0.8) return true;

    // Consider the ratio between learnt and original clauses
    if (learnts.size() > 0 && learnts.size() > 2 * nClauses()) return true;

    // Consider recent solver performance (conflict rate)
    if (conflictR > 1000 && learnts.size() > max_learnts * 0.8) return true;

    return false;
}
```

Figure 17. Example of LLM discovered heuristics of reduce condition