



# DynamicSAT: Dynamic Configuration Tuning for SAT Solving

Zhengyuan Shi ✉ 

The Chinese University of Hong Kong, Hong Kong  
National Center of Technology Innovation for  
EDA, Nanjing, China

Wentao Jiang ✉ 


Ningbo University, China  
National Center of Technology Innovation for  
EDA, Nanjing, China

Xindi Zhang<sup>1</sup> ✉ 

Key Laboratory of System Software (Chinese  
Academy of Sciences) and State Key Laborat-  
ory of Computer Science, Institute of Software,  
Chinese Academy of Science, Beijing, China

Jin Luo ✉ 


Peking University, Beijing, China  
National Center of Technology Innovation for  
EDA, Nanjing, China

Yun Liang ✉ 

Peking University, Beijing, China

Zhufei Chu ✉ 

Ningbo University, China

Qiang Xu<sup>1</sup> ✉ 

The Chinese University of Hong Kong, Hong Kong  
National Center of Technology Innovation for  
EDA, Nanjing, China

---

## Abstract

Boolean Satisfiability (SAT) problem serves as a foundation for solving numerous real-world challenges. As problem complexity increases, so does the demand for sophisticated SAT solvers, which incorporate a variety of heuristics tailored to optimize performance for specific problem instances. However, a major limitation persists: a configuration that performs well on one instance may lead to inefficiencies on others. While previous approaches to automatic algorithm configuration set parameters prior to runtime, they fail to adapt to the dynamic evolution of problem characteristics during the solving process. We introduce DynamicSAT, a novel SAT solver framework that dynamically tunes configuration parameters during solving process. By adjusting parameters on-the-fly, DynamicSAT adapts to changes arising from clause learning, elimination, and other transformations, thus improving efficiency and robustness across diverse SAT instances. We demonstrate that DynamicSAT achieves significant performance gains over the state-of-the-art solver on 2024 SAT Competition Benchmark.

**2012 ACM Subject Classification** Theory of computation → Constraint and logic programming; Mathematics of computing → Combinatorial optimization

**Keywords and phrases** Boolean satisfiability problem, configuration tuning, multi-armed bandit

**Digital Object Identifier** 10.4230/LIPIcs.CP.2025.34

**Supplementary Material** *Software (Source Code)*: <https://github.com/cure-lab/DynamicSAT>  
archived at `swb:1:dir:3c22d81c15345c48063f09c569fed3c286889034`

**Funding** This work was supported in part by the Hong Kong Research Grants Council (RGC) under Grant No. 14212422, 14202824, and C6003-24Y, in part by National Technology Innovation Center for EDA, China and in part by the NSFC under Grant No. 62274100, No. T2293700 and No. T2293701.

---

<sup>1</sup> The corresponding authors are: Xindi Zhang and Qiang Xu



## 1 Introduction

The Boolean Satisfiability (SAT) problem determines whether a given propositional logic formula can be satisfied, recognizing as the first NP-hard problem [8]. Its importance extends beyond theoretical computer science, as many real-world problems, including hardware design [29], planning [22], and scheduling [14], can be efficiently transformed into SAT instances in polynomial time. These problems are then tackled using SAT solvers, which provide a practical framework for addressing challenges across various domains. As the complexity of practical applications continues to grow, so does the demand for more sophisticated and efficient SAT-solving techniques.

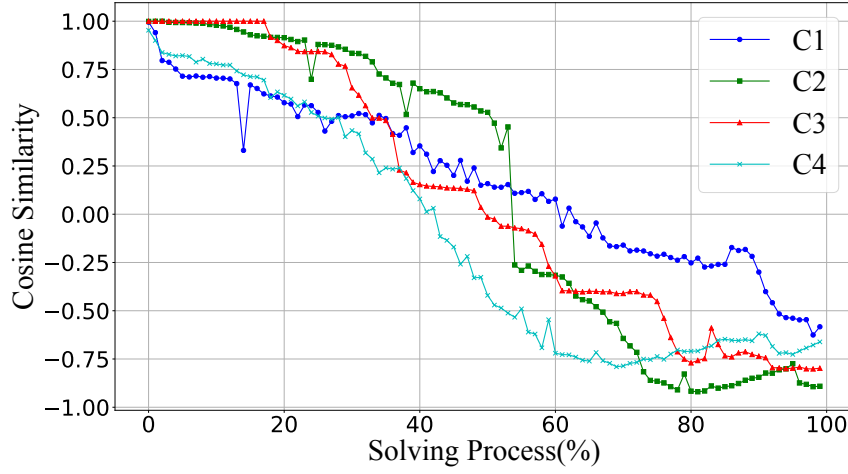
The mainstream SAT solvers opt for Conflict-Driven Clause-Learning (CDCL) [28] as the core searching algorithm. In CDCL framework, the problem instances is represented as Conjunctive Normal Form (CNF), consisting of a series of clauses. To enhance the efficiency of CDCL, substantial progress has been made in the SAT community, including advanced SAT solvers [31, 38, 5, 10] and their corresponding techniques [12, 3, 25, 36, 26]. For example, branching techniques [31, 25, 37] guide the solving process by selecting variables and branching orders. Besides, restart technique [12] has been introduced to periodically reset the solving process, helping solvers escape local optima and overcome stagnation in the search space. Clause management strategies [3] focus on simplifying the underlying problem by identifying and removing redundant or non-essential clauses, thus improving computational efficiency and guiding the solver toward viable solutions. These techniques form the backbone of modern SAT solvers, enabling them to handle increasingly complex and diverse applications.

Unfortunately, a well-known challenge remains: techniques that perform effectively on one class of instances may perform poorly on another [7, 24]. To address this variability, modern SAT solvers offer a range of configurable parameters, allowing users to customize heuristics and optimize solver performance for specific problem types. Therefore, algorithm configuration tuning becomes a promising direction in the SAT community, which automates the selection of suitable heuristic parameters, eliminating the need for manual updates. Representative techniques [20, 15, 9] determining the suitable configuration parameters prior to the solving process, have yielded advance across a wide range of problem solving.

Within the CDCL framework, the problem instance undergoes continuous transformation throughout the solving process. The initial CNF instance becomes increasingly dissimilar to its original state due to critical clause operations, such as clause elimination [13], which removes redundant clauses, and clause learning [34], which introduces new clauses into the instance. As illustrated in Figure 1, the cosine similarity between the intermediate instance and the initial instance decreases significantly over time. Consequently, this paper raises a critical question: *Does this divergence indicate that the initial configuration may no longer remain optimal during runtime, resulting in degraded solver performance?*

To address this question, we propose **DynamicSAT**, a novel solution that dynamically tunes the solver configuration during the solving process instead of maintaining the initial configuration throughout. Specifically, we model the configuration tuning process as a decision-making problem and incorporate a general reward function tied to solving efficiency. As a plug-in for modern SAT solvers, DynamicSAT adjusts arbitrary configuration parameters on-the-fly to maximize the reward metric. We employ a Multi-Armed Bandit (MAB) algorithm to make optimal decisions, where tuning configuration parameters corresponds to selecting action arms.

DynamicSAT is not merely a plug-in framework but a methodology to evaluate the necessity of dynamic tuning for SAT solving heuristics. We evaluate the effectiveness of solvers enhanced with DynamicSAT and compare them with baseline solvers. Specifically, we



■ **Figure 1** CNF changes during solving: We randomly select 4 problem instances (C1–C4). For each instance, we extract 100 intermediate CNFs at equal intervals throughout the solving process and generate their feature vectors using SATZilla [39]. The y-axis represents the cosine similarity between the feature vectors of each extracted intermediate CNF and the initial CNF.

tune the configurations of various modern heuristics, including chronological backtracking [32], clause vivification [23], and restart strategies [12]. The experiments are conducted on the 2024 SAT Competition benchmarks [19]. Our results demonstrate that DynamicSAT significantly improves solver efficiency, both in terms of the number of problems solved and the total solving time. Notably, our approach only adjusts the configurations of a single heuristic **compact** achieves 19.23% Par2 score reduction compared to Kissat-4.0.0, the champion of the 2024 SAT Competition [4].

We summarize the contributions of our work as follows.

- We observe and formally define the dynamic changes of problem instances during SAT solving, which may lead to the initial configuration becoming inefficient during the solving process.
- We propose a general Multi-Armed Bandit (MAB)-based approach to dynamically tune the configuration parameters of the solving algorithm during runtime. This method is versatile and can be applied to any configuration tuning of SAT solvers.
- We conduct experiments using various sets of tunable configuration parameters. The experimental results demonstrate that our approach can be easily implemented as a plug-in into baseline solver, achieving solving time reductions of up to 21.37%.

## 2 Related Work

### 2.1 Conflict-Driven Clause-Learning

The Boolean Satisfiability (SAT) problem is a fundamental problem in computer science and logic, which involves determining whether a given propositional formula is satisfiable. Specifically, the SAT problem is defined over a propositional formula in Conjunctive Normal Form (CNF). Let  $X$  be a set of propositional variables. A literal  $l$  is either a variable  $x \in X$  or its negation  $\neg x$ . The clause  $C_i$  is defined as the disjunction of literals and CNF formula  $F$  is a conjunction of  $m$  clauses.

$$F = C_1 \wedge C_2 \wedge \dots \wedge C_m \quad (1)$$

Solving the SAT problem is formulated as determining whether there exists an assignment  $A = l_1, l_2, \dots, l_n$  that leads the formula  $F$  to evaluate to logic-1 (*i.e.*,  $F$  is satisfied). If no such assignment exists, the formula  $F$  is said to be unsatisfiable.

Modern Boolean Satisfiability solvers primarily rely on Conflict-Driven Clause-Learning (CDCL) as their core algorithm [28], supplemented by various heuristics to efficiently address this NP-hard problem. In a CDCL solver, the problem instance is represented in Conjunctive Normal Form (CNF), which consists of a conjunction of clauses, where each clause is a disjunction of literals.

Over the past decades, numerous solving heuristics have been developed to accelerate the systematic exploration of potential solutions while leveraging learned conflicts to prune the search space. For example, clause vivification [23] reduces the search space by simplifying clauses through partial resolution, effectively strengthening constraints to eliminate redundancies. Chronological backtracking [32] revisits decisions in the reverse chronological order, avoiding expensive backtracking to arbitrary points in the decision tree. Besides, restart strategy [12] periodically restart the search process to escape local minima and improve convergence on hard instances. While these heuristics and their corresponding configuration parameters significantly enhance solver performance, determining the optimal configuration before or during SAT solving remains a critical challenge.

## 2.2 Algorithm Configuration in SAT Solver

Currently, SAT solvers employ a range of parameterized heuristics, with solver performance on a given instance being highly dependent on the choice of parameters [18]. Identifying an optimal parameter configuration for specific instances is critical to improving performance. One representative approach focuses on determining the best configuration for a set of CNF instances from a training set [17, 2, 16]. These methods often use diverse search strategies, such as iterated local search [17], genetic algorithms [2], and model-based optimization [16], to optimize configurations for a given set of instances. However, this search process requires time-intensive trials to evaluate configuration candidates, making these methods inefficient for application on a per-instance basis.

For instances with intrinsic differences, solving them under distinct configurations is generally more effective than using a single configuration across all instances. Instance-specific algorithm configuration (ISAC) [20] addresses this by clustering CNF instances based on their features using g-means clustering. Each cluster, composed of similar CNFs, then applies a search-based configuration strategy to identify optimal parameter settings. When encountering a new instance, ISAC assigns it to the nearest cluster and applies that cluster's configuration to the instance. Malitsky *et al.* [27] further enhance ISAC by introducing adaptive clustering, allowing the model to adjust clusters as new instances are added.

While these approaches have demonstrated performance gains by selecting appropriate parameters for individual instances, they do not address the dynamic changes that occur throughout the SAT solving process. As the characteristics of a CNF evolve during solving, the most effective strategy may also shift [7].

## 2.3 Multi-Armed Bandit

The Multi-Armed Bandit (MAB) problem [21] is a fundamental framework in reinforcement learning and decision-making under uncertainty. It models scenarios where an agent iteratively selects actions (termed “arms”,  $A = \{a_1, a_2, \dots, a_T\}$ ) to maximize cumulative rewards  $\sum r_t(a_t)$  while balancing exploration (gathering information about under-tested

arms) and exploitation (leveraging known high-reward arms). Common approaches to solving MAB problems include algorithms such as  $\epsilon$ -greedy, Upper Confidence Bound (UCB), and Thompson Sampling. These methods estimate the expected reward for each arm while efficiently managing the exploration-exploitation trade-off. For instance, UCB selects arms based on an optimistic estimate of their potential, while Thompson Sampling uses Bayesian inference to sample actions according to their probability of being optimal.

Since there is no explicit functional relationship between maximizing total expected reward and the selection of arms, MAB is particularly well-suited for addressing black-box optimization problems. This makes MAB-based approaches highly effective for dynamically adjusting configurations or strategies during the solving process, where accurately modeling the reward function is challenging. Previous work, such as Kissat-MAB [7], employs MAB to dynamically select between two state-of-the-art SAT solving heuristics: Variable State Independent Decaying Sum (VSIDS) and the Conflict History-Based (CHB) branching heuristic. Similarly, [24] formulates the decision of whether or not to trigger a solver reset as a MAB problem. However, these approaches focus narrowly on optimizing specific configurations rather than providing a generalized framework for handling arbitrary heuristics with minimal engineering effort.

In this work, we model configuration tuning as a MAB problem and propose a general reward and action formulation that supports a wide range of heuristic configuration parameters. This approach offers a versatile solution for dynamically optimizing SAT solver performance while minimizing the need for extensive engineering effort.

### 3 Methodology

#### 3.1 Problem Statment

The Boolean Satisfiability (SAT) problem can be defined over a propositional formula in conjunctive normal form (CNF). Let the initial CNF instance be denoted as Eq (1), where  $C$  represents a clause, and  $m$  is the number of clauses. We denote the CNF formula at variable decision step  $t$  as  $F_t$  and the initial CNF is  $F_0$ . During the execution of a Conflict-Driven Clause Learning (CDCL) SAT solver, the CNF formula evolves dynamically due to unit propagation and clause learning, clause elimination mechanisms.

**Unit Propagation.** At each step  $t$ , the partial assignment  $A_t$  simplifies  $F_t$  through Boolean constraint propagation [30]. Formally, let  $A_t = \{l_1, l_2, \dots, l_n\}$  be the literal assigned at step  $t$ . After unit propagation, the clauses satisfied by  $A_t$  (i.e.,  $\exists l \in C \cap A_t$ ) are discarded, while the unsolved clauses retain only unassigned literals:

$$F_t^{UP} = UnitProp(F_t, A_t) = \left\{ C \setminus \{\neg l \mid l \in A_t\} \mid C \in F_t, A_t \not\models C \right\} \quad (2)$$

**Clause Learning.** Following conflict analysis, the solver derives new clauses via resolution during conflict analysis [28]. Let  $\mathcal{L}_t$  denote the set of learned clauses added at step  $t$ . The augmented CNF  $F_t^{CL}$  becomes Eq (3). Each learned clause  $C_{learned} \in \mathcal{L}_t$  is logically implied by  $F_t$ , where  $F_t \models C_{learned}$ , but contains fewer literals than the clauses representing conflicts.

$$F_t^{CL} = F_t \cup \mathcal{L}_t \quad (3)$$

**Clause Elimination.** To maintain tractability, the solver periodically removes clauses deemed irrelevant by activity-based heuristics. Let  $\mathcal{E}_t$  denote the set of clauses eliminated at step  $t$ , which is selected by their activity scores [3]. The CNF after elimination becomes:

$$F_t^{CE} = F_t \setminus \mathcal{E}_t \quad (4)$$

Combining the above three mechanisms, the CNF at the next step  $t + 1$  is defined as below. This induces a non-stationary problem instance, where  $F_t$  structurally diverges from  $F_0$  through iterative clause modifications.

$$F_{t+1} = \text{UnitProp}(F_t, A_t) \cup \mathcal{L}_t \setminus \mathcal{E}_t \quad (5)$$

To better visualize this phenomenon, we quantify the structural divergence by analyzing the evolution of intermediate CNF features. We first randomly select 4 representative instances (C1–C4) from SAT competition benchmarks [19] and sample 100 intermediate CNF formulas  $F_t$  at uniform decision intervals (0–100% solving progress). We then employ SATZilla [39], a well-known tool has been successfully used in previous algorithm selection approaches [40, 35], to obtain feature vectors of  $F_t$  and  $F_0$ . Feature vectors represent CNF instances by incorporating various features, such as basic syntactic features (*e.g.*, clause/variable ratios), graph-based metrics (*e.g.*, nodes degree statistics of variable-clause graph) and behavioral indicators from short solver probes. We denote the feature vector  $h_t = \text{Norm}(\phi(F_t))$ , where  $\phi(\cdot)$  represents SATZilla’s feature mapping and  $\text{Norm}(\cdot)$  performs zero-mean normalization. The similarity between  $F_t$  and  $F_0$  is calculated as cosine similarity between  $h_t$  and  $h_0$ :

$$\text{sim}(F_t, F_0) = \frac{h_t \cdot h_0}{\|h_t\| \|h_0\|} \quad (6)$$

As shown in Figure 1, all four instances exhibit a decay of similarity, confirming that  $F_t$  becomes increasingly dissimilar to  $F_0$ . Consequently, this dynamic alteration can render the initial configuration inadequate for handling the evolving CNF in its intermediate state. Consequently, there is a clear motivation for dynamically adjusting the configuration parameters during the solving process.

## 3.2 Overview of DynamicSAT

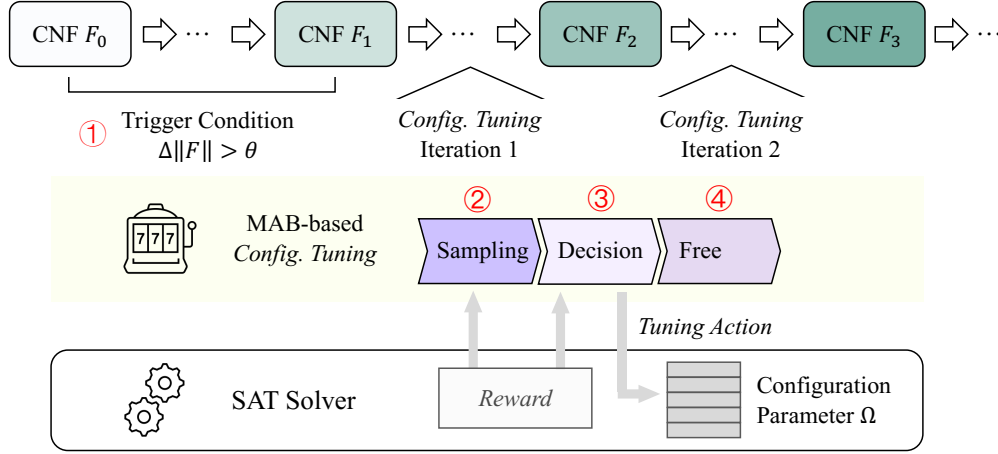
Given the structural divergence quantified by  $\text{sim}(F_t, F_0)$ , the initial configuration  $\Omega_0$  becomes suboptimal for  $F_t$ . We formalize on-the-fly parameter tuning as a decision-making problem where the action is how to adjust the parameters and the goal is to reducing solving time. To achieve this, we adopt the MAB algorithm to dynamically optimize solver behavior.

Figure 2 demonstrates the proposed MAB-based solving process, where the initial problem instance is denoted as  $F_0$  and the sampled intermediate instances are  $F_1, F_2, F_3, \dots$ . When the dynamic changes in the CNF instance meet a predefined trigger condition  $\Delta\|F\| \geq \theta$  (see Section 3.3.1), the MAB-based configuration tuning process is initiated, consisting of three stages: Sampling, Decision, and Free (see Section 3.4). The DynamicSAT serves as a plug-in within the SAT solver and repeats iteratively until the solver terminates to reduce solving time.

## 3.3 MAB Formulation

### 3.3.1 Trigger Condition

To determine when the problem instance has diverged sufficiently to necessitate parameter reconfiguration, we formalize a trigger condition based on the structural evolution of the CNF formula  $F_i$ .



■ **Figure 2** Overview of Dynamic Configuration Tuning.

As discussed in Section 3.1, the CDCL solver dynamically modifies  $F_i$  through unit propagation, clause learning and clause elimination, resulting in continuous additions and removals of clauses. These operations directly alter the formula's logical and topological properties, which we quantify using the number of added and removed clauses as metrics of structural change.

Let  $F_i$  to  $F_j$  denote CNF instances at steps  $i$  and  $j$ , respectively. We define:

- $\#add(F_i, F_j)$ : The number of clauses added to  $F_i$  to derive  $F_j$ . These include learned clauses generated during conflict analysis.
- $\#rm(F_i, F_j)$ : The number of clauses removed from  $F_i$  to derive  $F_j$  due to clause elimination and unit propagation.

The cumulative clause change between  $F_i$  and  $F_j$  is then defined as:

$$\Delta\|F_{i \rightarrow j}\| = \#add(F_i, F_j) + \#rm(F_i, F_j) \quad (7)$$

Given the current intermediate instance  $F_j$  and the instance from the previous timestamp  $F_i$ , the MAB-based configuration tuning approach is triggered if the clause changes  $\Delta\|F_{i \rightarrow j}\|$  exceed the threshold  $\theta$ . Once triggered, the timestamp is updated to instance  $F_j$ , ensuring that subsequent comparisons measure changes relative to  $F_j$ . It should be denoted that the initial change is defined as  $\Delta\|F_{0 \rightarrow j}\|$ , where  $F_0$  represents the initial instance.

### 3.3.2 Action Space

To enable adaptive configuration tuning in DynamicSAT, we formalize the action space as a set of discrete adjustments to solver parameters. Each action (*i.e.*, arm)  $a_i$  corresponds to a specific combination of parameter updates, which the MAB framework selects based on the evolving state  $F_i$ .

For Boolean parameters, which toggle specific heuristics on or off (e.g., enabling chronological backtracking), the tunable value  $v^{bool}$  is defined as a binary choice:

$$v^{bool} = \{0, 1\} \quad (8)$$

For integer parameters, which control thresholds or limits (e.g., backtracking depth), adjustments are constrained to three discrete actions increase its original value by 10%, decrease its by 10%, and keep it stable. It is denoted in Eq. (9), where  $p$  and  $q$  are the lower and upper bounds, respectively.

$$v^{int} = \begin{cases} \min(v^{int} * (1 + 10\%), q) \\ \max(v^{int} * (1 - 10\%), p) \\ v^{int} \end{cases} \quad (9)$$

Then, our DynamicSAT with multi-arms supports tuning multiple options to explore synergistic interactions. The action space  $\|a\|$  for tuning multiple parameters is the product of individual adjustments. For example, consider tuning the chronological backtracking heuristic [32], which provides Boolean option `--chrono` and int option `--chronolevels` in the integrated SAT solver. The combined action space for these parameters is  $\|a\| = 2 \cdot 3 = 6$ , where each arm in action  $a$  represents a unique decision of two parameters.

### 3.3.3 Reward

MAB strategies rely on a reward function to evaluate the performance of the selected actions. In our framework, we define the reward to reveal how the previous action benefit for the solving process. To enhance the flexibility of DynamicSAT, we adopt a general metric that evaluates the impact of configuration tuning on reducing the SAT solver's search space. Specifically, we assess the quality of recently learned clauses.

As reward, we utilize the recent Literals Blocks Distance (LBD) value [3], a metric effectively reflects the quality of learned clauses. We define the reward of action in timestamp  $i$  as  $r(a_i)$  in Eq. (10), where  $\text{Avg}(\text{LBD}(i - d, i))$  represents the average LBD value observed between timestamps  $i - d$  to  $i$  and  $d$  is the number of variable decision intervals in this period. Additionally,  $U$  is a constant number. Consequently, such reward function is designed to assign a higher reward when the arm selected in the previous evaluation round leads the solver to learn clauses of higher quality, thereby incentivizing actions that reduce the searching space during SAT solving.

$$r(a_i) = U - \text{Avg}(\text{LBD}(i - d, i)) \quad (10)$$

### 3.4 MAB Strategy

We opt for Upper Confidence Bound (UCB) algorithm [1] to implement our MAB-based configuration tuning. For each arm  $a$ , there are the following relevant parameters:

- $N(a)$  represents the number of times arm  $a$  is selected in the previous of evaluations.
- $\mathbb{E}(a) = \frac{\sum r(a)}{\#a}$  represents the expected reward of arm  $a$  over the previous of evaluations, where  $\#a$  is the total number of arm  $a_k$  selected.

The confidence upper bound for each action can be expressed as Eq. (11), where  $i$  is accumulated times of MAB decision in the current configuration tuning iteration until now.

$$UCB(a) = \mathbb{E}(a) + 2\sqrt{\frac{\ln(i)}{N(a)}} \quad (11)$$

The UCB algorithm selects the arm that maximizes the value on the right side of the above formula as the current action, and reduces its uncertainty as the number of selections increases, balancing exploration and exploitation.



■ **Algorithm 1** Pseudo-code of implementing DynamicSAT in variable decision function.

---

**Input:** The number of variable decision times  $t$   
 // Main program of variable decision

```

1 ...
  // DynamicSAT code starts below:
2 Define the interval of MAB decision making as  $d$ 
  // 1. Condition to trigger the decision process
3 if  $\Delta\|F_{i \rightarrow j}\| \geq \theta$  then
4   trigger  $\leftarrow$  True // Set trigger to True
5   no_sampling  $\leftarrow \mu$  // Set sampling frequency
6   no_decision  $\leftarrow \lambda$  // Set decision frequency
7    $i \leftarrow 0$  // Initialize of MAB decisions counter
8 end
9 else
10  trigger  $\leftarrow$  False // Keep trigger to False
11 end
  // Process when triggered
12 if trigger then
  // 2. Sampling Stage
13  while no_sampling > 0 and  $t\%d == 0$  do
14    Get  $r(a_i)$  // Get reward for previous action
15    Update  $\mathbb{E}(a_i)$  // Update expected value in MAB
16    no_sampling  $\leftarrow$  no_sampling - 1
17     $i \leftarrow i + 1$ 
18     $a_i \leftarrow$  Randomly select action // Randomly select for exploration
19    Update  $\Omega$  with action  $a_i$  // Update the solving configuration
20  end
  // 3. Decision Stage
21  while no_sampling == 0 and no_decision > 0 and  $t\%d == 0$  do
22    Get  $r(a_i)$  // Get reward for previous action
23    Update  $\mathbb{E}(a_i)$  // Update expected value in MAB
24    no_decision  $\leftarrow$  no_decision - 1
25     $i \leftarrow i + 1$ 
    // Make decision based on UCB value
26     $a_i \leftarrow \arg \max(\text{UCB})$  // See Eq. (11)
27    Update  $\Omega$  with action  $a_i$  // Update the solving configuration
28  end
  // 4. Free Stage
29  if no_sampling == 0 and no_decision == 0 then
30    trigger  $\leftarrow$  False
31    Clear UCB value
32  end
33 end
  
```

---

In DynamicSAT, we divide each configuration tuning iteration into three stages: Sampling, Decision and Free, as shown in Algorithm 1. This approach ensures a balance between exploration and exploitation while maintaining computational efficiency and flexibility. We define the hyperparameters  $\theta$  represents the formula change threshold,  $d$  is the intervals of MAB decision making and  $\mu$  is the sampling frequency. In the default setting, we assign  $\mu = 1000$ ,  $\theta = 30\%$  and  $d = 100$ , which are explored in Section 4.2.

Once the trigger condition satisfied (Line 3), the MAB process initiates exploration by dynamically selecting actions through random sampling (Line 18). For each action  $a_i$ , the reward  $r(a_i)$  is collected according to Eq. (10) (Line 14) and expected value  $\mathbb{E}(a_i)$  is iteratively updated (Line 15). This stage prioritizes exploration to gather sufficient data for the current intermediate instance, ensuring the policy avoids premature convergence to suboptimal configurations.

Then, once exploration concludes ( $no\_sampling=0$ ), the algorithm shifts to exploitation. Here, the updated policy guides the selection of action arms to maximize the reward, using UCB value (Line 26, Eq. (11)). Configuration parameters  $\Omega$  are adjusted based on the arm with the highest UCB value.

After both sampling and decision, the MAB-based process is disabled (Line 29). The solver keeps the current optimal configuration until the next trigger condition is met. Besides, the UCB value of each action is cleared before the next iteration.

## 4 Experiments

### 4.1 Experimental Settings

Our DynamicSAT can be seamlessly integrated into the modern SAT solvers with minimal engineering effort, while also supporting configuration tuning for arbitrary heuristics. We opt for Kissat-4.0.0 (Kissat-sc24), the winner of SAT Competition 2024 [4] as the baseline solver. To demonstrate the effectiveness of DynamicSAT, we introduce specific variants (named **DSAT-[heuristic]**) that integrate with the baseline solver and allow for the dynamic tuning of multiple heuristic options. For instance, the variant named **DSAT-chrono** refers to the solver where the options related to chronological backtracking [32] are dynamically tunable during runtime. Specifically, the baseline solver provide two options: `--chrono` and `--chronolevels`.

In our MAB setting, sampling occurs at intervals of every  $p$  decision rounds, with the decision phase triggered when the number of processed clauses reaches  $\theta$  of the total clauses in the original CNFs. For each MAB iteration, DynamicSAT conducts sampling for  $s$  times before making decisions based on the UCB value. To ensure consistency, we use the same random seed across all experiments, eliminating the effect of randomness. All experiments are conducted on an Intel(R) Xeon(R) CPU E7-8860 v4 @ 2.20GHz with 128GB memory. The timeout is set to 5,000 seconds of CPU time.

We test the effectiveness of DynamicSAT on the 2024 SAT Competition Benchmark [19] and evaluate using three metrics.

- **Problem Solved (#Solved)**: The number of problem instances that can be solved within time limit. A higher value indicates better solver performance.
- **Solving Time of Solvable Instance (Time)**: The average runtime for successfully solved instances by baseline solver, which investigate whether DynamicSAT negatively impacts the performance on these relative easy instances. A lower value indicates better solver performance.

■ **Table 1** The results of hyperparameters exploration.

Initial Sampling Frequency			Clause Change Threshold			Decision Interval		
$\mu$	#Solved	Par2	$\theta$	#Solved	Par2	$d$	#Solved	Par2
10	308	2,852.60	10%	300	3,038.82	10	305	2,841.36
100	303	2,953.52	<b>30%</b>	<b>305</b>	<b>2,910.31</b>	<b>100</b>	<b>310</b>	<b>2,721.20</b>
<b>1,000</b>	<b>310</b>	<b>2,824.76</b>	50%	303	2,964.96	1,000	308	2,751.27
10,000	305	2,875.46	100%	302	2,966.27	10,000	308	2,760.65

- **Average Par2 Score (Par2):** The Par2 score sums the solving time for successfully solved instances and applies a penalty of twice the time limit for instances that timeout. In our experiments, we report the average Par2 score. A lower Par2 score indicates better solver performance.

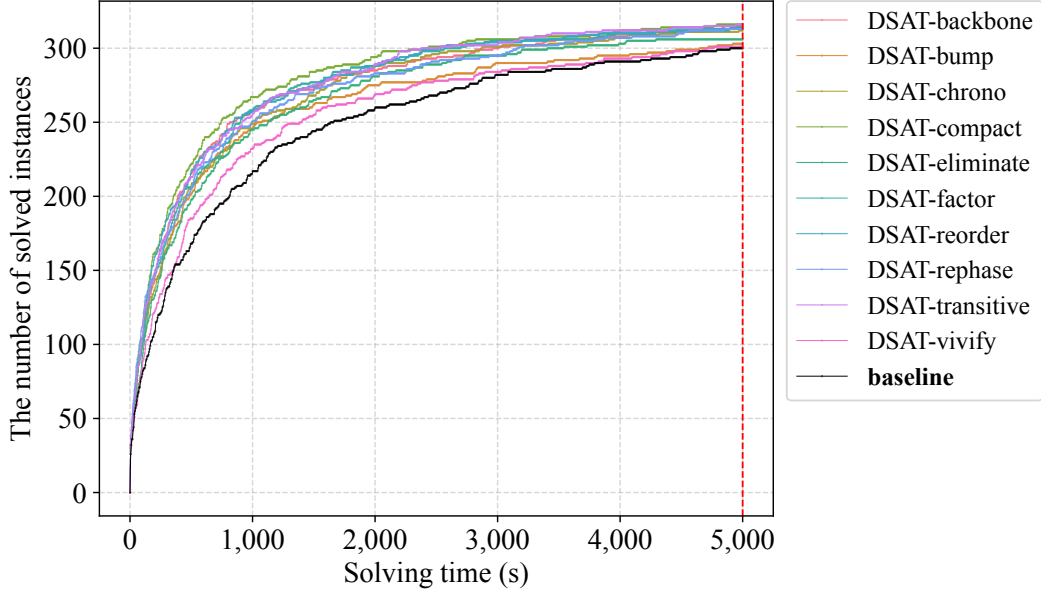
## 4.2 Hyperparameter Analysis of DynamicSAT

DynamicSAT’s performance hinges on three critical hyperparameters: the initial sampling frequency ( $\mu$ ), the clause change threshold ( $\theta$ ), and the decision interval ( $d$ ). We analyze their impact using DSAT-chrono on the the benchmark, with results summarized in Table 1. All experiments fix two hyperparameters while varying the third, with results summarized in Table 1.

**Initial Sampling Frequency.** The initial sampling frequency  $\mu$  determines how many arms (actions) are tested during the sampling stage when the MAB framework is first activated. A higher  $\mu$  increase early-stage exploration, refining reward estimates  $E(a)$ , but increases computational overhead. From Table 1, we observe that  $\mu = 1,000$  achieves the best performance: it solves 310 instances (vs. 308 for  $\mu = 10$ ) with the lowest Par2 score 2,824.76. Smaller values ( $\mu = 10$  and  $\mu = 100$ ) under-explore, leading to suboptimal policy convergence, while  $\mu = 10,000$  over-samples, wasting runtime on redundant exploration. Therefore, moderate early-stage sampling benefits reward estimation accuracy without sacrificing exploitation efficiency, where we assign  $\mu = 1,000$  as the default setting.

**Clause Change Threshold.** The clause change threshold  $\theta$  triggers MAB-based configuration tuning when the normalized divergence  $\Delta\|F_{i \rightarrow j}\| \geq \theta$ . First, a lower  $\theta = 10\%$  results in increasing sensitivity to formula dynamics but risks overreacting to noise. For instance, the variant with  $\theta = 10\%$  demonstrates the worst performance, solving only 300 instances. Second, a higher threshold  $\theta = 50\%$  or  $\theta = 100\%$  is too insensitive to changes, resulting in suboptimal performance. Third, at  $\theta = 30\%$ , DynamicSAT solves 305 instances with a Par2 of 2,910.31, outperforming the others. This suggests  $\theta = 30\%$  optimally filters transient clause fluctuations while capturing critical structural shifts.

**Decision Interval.** The decision interval  $d$  indicates the number of variable decision steps (*i.e.*, the times entry main program of variable decision in SAT solver) between consecutive MAB decision. Since our reward function depends on the quality of learned clauses over a period, shorter intervals may result in insufficient clause learning, while longer intervals risk delayed adaptation. From Table 1,  $d = 100$  achieves the best Par2 score (2,721.20) while solving 310 instances. In contrast,  $d = 10$  degrades performance by 4.42% in Par2 score due to the frequent reward access, while  $d = 1,000$  and  $d = 10,000$  causes stale policies, resulting in suboptimal performance. Therefore,  $d = 100$  is adopted as the default in the subsequent experiments.



■ **Figure 3** Number of solved instances as a function of solving time.

### 4.3 Effect of DynamicSAT

To quantify the impact of integrating DynamicSAT on various in-processing heuristics<sup>1</sup>, we choose 10 heuristic strategies randomly and evaluate the corresponding DynamicSAT variants. The complete list of tunable heuristics and their corresponding options is provided in Table 7 in Appendix A.1. Table 2 compares their performance against the baseline solver on the 2024 SAT Competition Benchmark. It should be noted that the runtime includes DynamicSAT, even though it occupies a small portion of the overall time (see discussion in Appendix A.3). We conclude three observations from the results.

First, the results reveal that all 10 solver variants powered by our proposed DynamicSAT consistently outperform the baseline solver across all metrics. On average, DynamicSAT achieves a 3.63% increase in #Solved (310.90 vs. 300), 32.30% solving time reduction (568.49 vs. 839.70) on solvable instances and 13.42% Par2 reduction (2,709.68 vs. 3,129.77), confirming the robustness of our approach across diverse heuristics. Besides, Figure 3 further illustrates the superiority of DynamicSAT: the curves of all 10 variants rise steadily over time, consistently surpassing the baseline’s plateau.

Second, the most significant improvements arise from heuristics directly interacting with the clause database. Notably, the top-performing variant **DSAT-compact** solve 5.33% more instances than the baseline, with a 19.23% reduction in Par2 score, demonstrating its superior efficiency and effectiveness. Similarly, **DSAT-transitive** also achieves 5.33% more instances solved. These strategies adaptively prune redundant clauses and simplify implication graphs during critical phases of structural divergence [6]. Since these heuristics are more sensitive to the structural CNF changes, tuning their corresponding parameters ensures optimal configuration.

<sup>1</sup> We do not adjust the pre-processing heuristics (e.g., **fastel** and **lucky**), as they are executed only once before the solving process.

■ **Table 2** Performance comparison between baseline and DynamicSAT.

	#Solved	Impv.	Time	Red.	Par2	Red.
baseline	300		839.70		3,129.77	
DSAT-backbone	315	5.00%	473.87	43.57%	2,616.12	16.41%
DSAT-bump	303	1.00%	715.23	14.82%	2,916.44	6.82%
DSAT-chrono	312	4.00%	595.24	29.11%	2,691.16	14.01%
DSAT-compact	<b>316</b>	<b>5.33%</b>	<b>444.16</b>	<b>47.10%</b>	<b>2,528.06</b>	<b>19.23%</b>
DSAT-eliminate	306	2.00%	687.71	18.10%	2,828.60	9.62%
DSAT-factor	314	4.67%	500.34	40.41%	2,618.74	16.33%
DSAT-reorder	313	4.33%	471.75	43.82%	2,616.13	16.41%
DSAT-rephase	313	4.33%	540.49	35.63%	2,678.69	14.41%
DSAT-transitive	<b>316</b>	<b>5.33%</b>	471.89	43.80%	2,574.10	17.75%
DSAT-vivify	301	0.33%	784.17	6.61%	3,028.80	3.23%
<b>Avg.</b>	<b>310.9</b>	<b>3.63%</b>	<b>568.49</b>	<b>32.30%</b>	<b>2,709.68</b>	<b>13.42%</b>

■ **Table 3** Performance comparison between baseline and DynamicSAT on SAT instances.

Solver	#Solved	Impv.	Time	Red.	Par2	Red.
baseline	149		660.49		1,462.65	
DSAT-backbone	154	3.36%	376.50	43.00%	987.05	32.52%
DSAT-bump	149	0.00%	488.71	26.01%	1,210.50	17.24%
DSAT-chrono	152	2.01%	527.31	20.16%	1,071.29	26.76%
DSAT-compact	155	4.03%	353.18	46.53%	<b>849.92</b>	<b>41.89%</b>
DSAT-eliminate	150	0.67%	630.54	4.53%	1,223.24	16.37%
DSAT-factor	155	4.03%	356.00	46.10%	885.90	39.43%
DSAT-reorder	153	2.68%	365.93	44.60%	981.02	32.93%
DSAT-rephase	153	2.68%	366.75	44.47%	982.87	32.80%
DSAT-transitive	<b>156</b>	<b>4.70%</b>	<b>347.25</b>	<b>47.43%</b>	853.90	41.62%
DSAT-vivify	153	2.68%	537.54	18.61%	1,122.96	23.22%
<b>Avg.</b>	<b>153.0</b>	<b>2.68%</b>	<b>434.97</b>	<b>34.14%</b>	<b>1,016.87</b>	<b>30.48%</b>

Third, heuristics focusing on localized variable operations exhibit marginal gains, where DSAT-vivify and DSAT-bump only reduce Par2 score by 3.23% and 6.82%, respectively. Such strategies, which optimize literal-level properties or activity scores, lack systemic alignment with the solver’s evolving state [25, 23]. Their localized impact and insensitivity to structural CNF shifts prevent them from benefiting meaningfully from dynamic adaptation. Moreover, we further analyze that DSAT-vivify performs particularly worse on UNSAT cases, as detailed in Section 4.4.

Therefore, our experiments highlight that DynamicSAT excels when the tunable heuristics interact with global structures but struggles with contextually isolated optimizations. Our DynamicSAT also supports tuning more options with complex interactions, which is considered as the future work and briefly discussed in Appendix A.2.

■ **Table 4** Performance comparison between baseline and DynamicSAT on UNSAT instances.

Solver	#Solved	Impv.	Time	Red.	Par2	Red.
baseline	151		1,016.53		1,778.76	
DSAT-backbone	161	6.62%	569.94	43.93%	1,003.39	43.59%
DSAT-bump	154	1.99%	938.74	7.65%	1,510.69	15.07%
DSAT-chrono	160	5.96%	662.28	34.85%	1,102.08	38.04%
DSAT-compact	<b>161</b>	<b>6.62%</b>	<b>533.93</b>	<b>47.48%</b>	<b>925.36</b>	<b>47.98%</b>
DSAT-eliminate	156	3.31%	744.12	26.80%	1,285.17	27.75%
DSAT-factor	159	5.30%	642.77	36.77%	1,109.66	37.62%
DSAT-reorder	160	5.96%	576.17	43.32%	1,009.37	43.25%
DSAT-rephase	160	5.96%	711.94	29.96%	1,159.21	34.83%
DSAT-transitive	<b>160</b>	<b>5.96%</b>	<b>594.88</b>	<b>41.48%</b>	1,033.06	41.92%
DSAT-vivify	<u>148</u>	<u>-1.99%</u>	<u>1,027.53</u>	<u>-1.08%</u>	<u>1,869.57</u>	<u>-5.11%</u>
<b>Avg.</b>	<b>157.9</b>	<b>4.57%</b>	<b>700.23</b>	<b>31.12%</b>	<b>1,200.76</b>	<b>32.49%</b>

#### 4.4 Failure Case Analysis

We investigate the reasons behind the the negative impacts observed with **DSAT-vivify** in this subsection. To further dissect the efficacy of DynamicSAT, we analyze its performance separately on satisfiable (SAT) and unsatisfiable (UNSAT) instances from the 2024 SAT Competition Benchmark. This distinction is critical, as SAT and UNSAT instances impose fundamentally different demands on solvers: SAT solutions require efficient search space navigation, while UNSAT proofs depend on conflict analysis and clause learning [33].

For satisfiable instances, DynamicSAT variants demonstrate substantial efficiency gains. In Table 3, adjusting heuristics **compact** achieves 41.89% Par2 reduction, while **DSAT-transitive** reduces the average solving time by 47.43% compared to the baseline. Although some variants, such as **DSAT-bump** and **DSAT-eliminate**, show only minor improvements in the number of solved instances, they remain effective by achieving significant reductions in both Par2 scores and solving time. For example, **DSAT-bump** does not solve any additional instances but still reduces the Par2 score by 17.24% and the solving time by 26.01%.

In contrast, unsatisfiable instances reveal a more nuanced performance landscape. While most DynamicSAT variants still outperform the baseline, variable-centric heuristics like **DSAT-vivify** exhibit limited gains, which aligns with the results in Section 4.3. To be specific, the worst-performing variant, **DSAT-vivify**, solves 1.99% fewer instances than the baseline solver and shows negative gains in solving time and Par2 score, with increases of 1.08% and 5.11%, respectively.

We attribute such limited efficacy to the inherent optimizations on the corresponding heuristic. As reported in the solver specification [4], the optimized and simplified vivification is one of the most significant update appeared in the baseline solver. This suggests that human expertise has largely saturated vivification’s optimization potential, while dynamic tuning it may gain diminish for highly engineered tactics. To further prove our hypothesis, we integrate DynamicSAT into the kissat solver without meticulous optimization on vivification, using the version **sc2022-bulky** [11] submitted to SAT Competition 2022 (denoted as **kissat-sc22**). The intergated variant is named as **DSAT-vivify-sc22**. The following experiments are conducted on the 2024 SAT Competition Benchmark under the same environment and hyperparameters as described earlier.

■ **Table 5** Performance comparison between kissat-sc22, kissat-sc24 and DSAT-vivify-sc22.

Solver	#Solved	Impv.	Time	Red.	Par2	Red.
Kissat-sc22	294		944.95		3,344.54	
Kissat-sc24	300	2.04%	1,087.49	-15.08%	3,129.77	<b>6.42%</b>
DSAT-vivify-sc22	295	0.34%	787.34	16.68%	3,155.88	<b>5.64%</b>

Table 5 summarizes the results. On the one hand, the variant with DynamicSAT **DSAT-vivify-sc22** achieves a 16.68% solving time reduction for solvable instances and a 5.64% reduction in Par2 score across the entire benchmark compared to **Kissat-sc22**, which highlights the effectiveness of DynamicSAT. On the other hand, the upgrade from **Kissat-sc22** to **Kissat-sc24** achieves a 2.04% increase in #Solved and a 6.42% reduction in Par2 score. Although **DSAT-vivify-sc22**, which simply combines DynamicSAT into **Kissat-sc22**, does not significantly improve #Solved, it achieves a 5.64% Par2 reduction, which is comparable to the improvement seen in **Kissat-sc24**. Our experimental results underscore the complementary roles of human expertise and automated adaptation in solver design, demonstrating that DynamicSAT can significantly enhance performance even without meticulous heuristic-specific optimization.

## 4.5 Instance Family Analysis

We present a breakdown analysis of instance families alongside the corresponding solver variants. Detailed results are provided in Table 8 in the Appendix A.4. Below, we discuss the performance trends observed for representative tuning heuristics.

First, **DSAT-backbone** and **DSAT-transitive** are the variants that support dynamic tuning of binary clause-related heuristics. The former focuses on analyzing binary clause backbones, while the latter applies transitive reduction to binary clauses. In families with a high density of binary clauses, such as Hamiltonian, Maxsat, and Miter, these heuristics yield substantial improvements. For instance, in the Hamiltonian family, the baseline solver achieves an average Par2 score of 404.76. **DSAT-backbone** reduces this score to 243.36 (a 39.88% improvement), and **DSAT-transitive** further lowers it to 198.25 (a 51.02% improvement). In contrast, **DSAT-chrono** only achieves an 8.93% Par2 reduction on Hamiltonian instances, and **DSAT-eliminate** even increases the Par2 score by 70.96%.

Second, for Cryptography instances, the baseline Par2 score is 1,310.32, and most tuning approaches yield marginal or negative improvements – for example, **DSAT-chrono** worsens performance by 57.13% and **DSAT-rephase** increases the Par2 score by 9.50%. Surprisingly, **DSAT-backbone** reduces the Par2 score by 68.39% on this instance family. Although cryptography instances do not present an abundance of explicit binary clauses, many variable relationships can be modeled as equality or inequality conditions that reflect the XOR structure. During the clause learning process, the solver adds numerous binary clauses representing these conditions. By dynamically tuning the backbone heuristic, the solver efficiently identifies and exploits these critical binary relationships, which leads to significant performance gains. This result highlights the adaptive capability of DynamicSAT to tailor tuning strategies to the dynamic properties of the instance family.

Third, our further experiments with a separated analysis across instance families align with the results in Section 4.4, where the variant **DSAT-vivify** does not significantly outperform the baseline. While **DSAT-vivify** reduces the Par2 score across most instance families, it struggles with solving Random Circuit, Miter, and Hamiltonian instances. Specifically, it even increases Par2 score on Random Circuit problems by 133.37%.



As a result, most DynamicSAT variants achieve remarkable performance on a broad range of problems. Moreover, our results clearly expose the specific instance families where DynamicSAT variants are less effective. In future work, we aim to refine our heuristics and further adapt the DynamicSAT framework to better address the limitations identified in these challenging instance families.

## 5 Conclusion

This paper introduces DynamicSAT, a framework for dynamically tuning configuration parameters on-the-fly within a SAT solver. Our work builds on the observation that CNF instances undergo significant changes during the solving process, resulting in a substantial difference between intermediate instances and the initial problem formulation. To address this, DynamicSAT employs a multi-armed bandit (MAB)-based agent to explore optimal configurations that adapt to the evolving characteristics of CNF instances. By integrating DynamicSAT into a baseline SAT solver, we achieve Par2 reductions of up to 19.23% on the 2024 SAT competition benchmarks. DynamicSAT is flexible and can tune arbitrary configurations of various SAT solvers with minimal engineering effort. We believe that this approach could become an efficient and general tool for advancing SAT solving frameworks.

As future work, several avenues can be explored to enhance DynamicSAT. First, it would be beneficial to investigate more effective settings in our MAB agent, including exploring alternative reward functions and considering other MAB algorithms (e.g.  $\epsilon$ -greedy and Thompson Sampling). Additionally, we should explore more effective approaches to simultaneously tune multiple heuristics, leveraging their synergistic effects to further enhance solver performance across diverse instance families. Finally, we aim to generalize the proposed DynamicSAT framework and integrate it into other modern SAT solvers.

---

## References

- 1 Rajeev Agrawal. Sample mean based index policies by  $o(\log n)$  regret for the multi-armed bandit problem. *Advances in applied probability*, 27(4):1054–1078, 1995.
- 2 Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 142–157, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-04244-7\_14.
- 3 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Twenty-first international joint conference on artificial intelligence*. Citeseer, 2009.
- 4 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*, volume B-2024-1 of *Department of Computer Science Report Series B*, pages 8–10. University of Helsinki, 2024.
- 5 Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.
- 6 Ronen I Brafman. A simplifier for propositional formulas with many binary clauses. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(1):52–59, 2004. doi:10.1109/TSMCB.2002.805807.
- 7 Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. Combining vsids and chb using restarts in sat. In *27th International Conference on Principles and Practice of Constraint Programming*, 2021.



- 8 Stephen A Cook. The complexity of theorem-proving procedures. In *Logic, automata, and computational complexity: The works of Stephen A. Cook*, pages 143–152. 2023. doi:10.1145/3588287.3588297.
- 9 Stefan Falkner, Marius Lindauer, and Frank Hutter. Spysmac: Automated configuration and performance analysis of sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 215–222. Springer, 2015. doi:10.1007/978-3-319-24318-4\_16.
- 10 ABKFM Fleury and Maximilian Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. *SAT COMPETITION*, 2020:50, 2020.
- 11 Armin Biere Mathias Fleury. Gimsatul, isasat, kissat. *SAT COMPETITION*, 10, 2022.
- 12 Carla P Gomes, Bart Selman, Henry Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98(1998):431–437, 1998.
- 13 Marijn Heule, Matti Järvisalo, Florian Lonsing, Martina Seidl, and Armin Biere. Clause elimination for sat and qsat. *Journal of Artificial Intelligence Research*, 53:127–168, 2015. doi:10.1613/JAIR.4694.
- 14 Andrei Horbach. A boolean satisfiability approach to the resource-constrained project scheduling problem. *Annals of Operations Research*, 181:89–107, 2010. doi:10.1007/S10479-010-0693-2.
- 15 Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*, pages 507–523. Springer, 2011. doi:10.1007/978-3-642-25566-3\_40.
- 16 Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-25566-3\_40.
- 17 Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, October 2009. doi:10.1613/JAIR.2861.
- 18 Frank Hutter, Marius Lindauer, Adrian Balint, Sam Bayless, Holger Hoos, and Kevin Leyton-Brown. The configurable sat solver challenge (cssc). *Artificial Intelligence*, 243:1–25, 2017. doi:10.1016/j.artint.2016.09.006.
- 19 Markus Iser and Christoph Jabs. Global Benchmark Database. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:10, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SAT.2024.18.
- 20 Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac–instance-specific algorithm configuration. In *ECAI 2010*, pages 751–756. IOS Press, 2010. doi:10.3233/978-1-60750-606-5-751.
- 21 Michael N Katehakis and Arthur F Veinott Jr. The multi-armed bandit problem: decomposition and computation. *Mathematics of Operations Research*, 12(2):262–268, 1987. doi:10.1287/MOOR.12.2.262.
- 22 Henry Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *IJCAI*, volume 99, pages 318–325, 1999. URL: <http://ijcai.org/Proceedings/99-1/Papers/047.pdf>.
- 23 Chu-Min Li, Fan Xiao, Mao Luo, Felip Manyà, Zhipeng Lü, and Yu Li. Clause vivification by unit propagation in cdcl sat solvers. *Artificial Intelligence*, 279:103197, 2020. doi:10.1016/J.ARTINT.2019.103197.
- 24 Chunxiao Li, Charlie Liu, Jonathan Chung, Piyush Jha, Vijay Ganesh, et al. A reinforcement learning based reset policy for cdcl sat solvers. *arXiv preprint arXiv:2404.03753*, 2024.
- 25 Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers. In *Hardware*

- and Software: Verification and Testing: 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, *Proceedings 11*, pages 225–241. Springer, 2015. doi:10.1007/978-3-319-26287-1\_14.
- 26 Hongduo Liu, Peng Xu, Yuan Pu, Lihao Yin, Hui-Ling Zhen, Mingxuan Yuan, Tsung-Yi Ho, and Bei Yu. Neuroselect: Learning to select clauses in sat solvers. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6, 2024. doi:10.1145/3649329.3656250.
  - 27 Yuri Malitsky, Deepak Mehta, and Barry O’Sullivan. Evolving instance specific algorithm configuration. In *Proceedings of the International Symposium on Combinatorial Search*, volume 4, pages 133–140, 2013. doi:10.1609/SOCS.V4I1.18296.
  - 28 Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 133–182. ios Press, 2021. doi:10.3233/FAIA200987.
  - 29 João P Marques-Silva and Karem A Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Annual Design Automation Conference*, pages 675–680, 2000. doi:10.1145/337292.337611.
  - 30 David A McAllester. Truth maintenance. In *AAAI*, volume 90, pages 1109–1116, 1990. URL: <http://www.aaai.org/Library/AAAI/1990/aaai90-164.php>.
  - 31 Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001. doi:10.1145/378239.379017.
  - 32 Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In *Theory and Applications of Satisfiability Testing–SAT 2018: 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings 21*, pages 111–121. Springer, 2018. doi:10.1007/978-3-319-94144-8\_7.
  - 33 Chanseok Oh. Between sat and unsat: the fundamental difference in cdcl sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 307–323. Springer, 2015. doi:10.1007/978-3-319-24318-4\_23.
  - 34 Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers as resolution engines. *Artificial intelligence*, 175(2):512–525, 2011. doi:10.1016/J.ARTINT.2010.10.002.
  - 35 Hadar Shavit and Holger H Hoos. Revisiting satzilla features in 2024. In *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, pages 27–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.
  - 36 Zhengyuan Shi, Min Li, Yi Liu, Sadaf Khan, Junhua Huang, Hui-Ling Zhen, Mingxuan Yuan, and Qiang Xu. Satformer: Transformer-based unsat core learning. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–4. IEEE, 2023. doi:10.1109/ICCAD57390.2023.10323731.
  - 37 Zhengyuan Shi, Hongyang Pan, Sadaf Khan, Min Li, Yi Liu, Junhua Huang, Hui-Ling Zhen, Mingxuan Yuan, Zhufei Chu, and Qiang Xu. Deepgate2: Functionality-aware circuit representation learning. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2023. doi:10.1109/ICCAD57390.2023.10323798.
  - 38 Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.
  - 39 Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008. doi:10.1613/JAIR.2490.
  - 40 Lin Xu, Frank Hutter, Jonathan Shen, Holger H Hoos, and Kevin Leyton-Brown. Satzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge*, pages 57–58, 2012.

## A Appendix

### A.1 Tunable Options in Kissat Solver

See Table 7. For each heuristic, our DynamicSAT tunes the corresponding configurations, which include all the relevant options in the baseline solver.

### A.2 Effect of DynamicSAT on Multiple Configuration Tuning

We employ DynamicSAT for the simultaneous tuning of options across multiple heuristics. Given the large number of possible combinations of tunable heuristics, we focus on two representative sets:

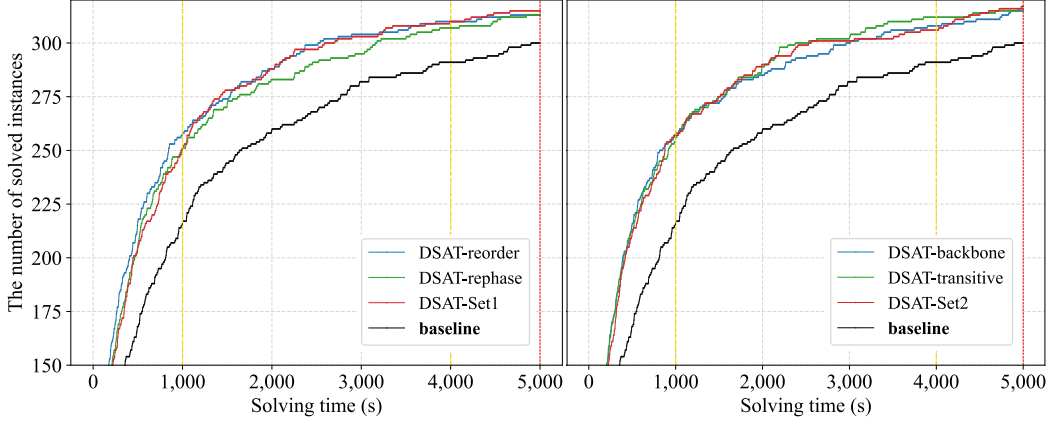
- **DSAT-Set1.** Combines variable decision heuristics **reorder** and **rephase**. These heuristics exhibit high interdependence in variable decision-making, as **reorder** alters the sequence of decision variables while **rephase** directly impacts the polarity choices during branching.
- **DSAT-Set2.** Integrates heuristics related to binary clause simplification. The **backbone** heuristic targets stable variable assignments in binary clauses, while the **transitive** heuristic simplifies binary clauses through transitive reduction.

■ **Table 6** Performance of DynamicSAT with multiple configuration tuning.

Solver	#Solved	Impv.	Time	Red.	Par2	Red.
baseline	300		839.70		3,129.77	
DSAT-reorder	313	4.33%	471.75	43.82%	2,616.13	16.41%
DSAT-rephase	313	4.33%	540.49	35.63%	2,678.69	14.41%
DSAT-Set1	315	5.00%	542.36	35.41%	2,616.70	16.39%
DSAT-backbone	315	5.00%	473.87	43.57%	2,616.12	16.41%
DSAT-transitive	316	5.33%	471.89	43.80%	2,574.10	17.75%
DSAT-Set2	317	0.63%	478.80	42.98%	2,586.11	17.37%

Table 6 compares the performance of these sets against their individual components and the baseline. We observe that jointly tuning interdependent heuristics improves the number of solved instances. Specifically, **DSAT-Set1** solves 315 instances, outperforming the individual tuning of single heuristic (**DSAT-reorder** and **DSAT-rephase**), which solve 313 instances each. However, the results also indicate that jointly tuning heuristics does not consistently show a positive synergistic effect on Par2 reduction and solving time reduction compared to tuning individual heuristics.

To further investigate the reasons behind the observed performance patterns, we analyze the number of solved instances as a function of a partial range of solving time. Figure 4 reveals that while **DSAT-Set1** ultimately solves more instances than individual heuristics, its early-stage inefficiency limits overall gains. For solving time  $\leq 1,000$ s, standalone heuristics (**DSAT-reorder**, **DSAT-rephase**) resolve more instances. For example, **DSAT-reorder** solves 258 instances within the 1,000s time limit, while **DSAT-Set1** only solves 251 instances. Beyond 4,000 seconds, **DSAT-Set1** overtakes individual heuristics. This dichotomy stems from the insufficient exploration in the MAB-based framework under tight time constraints. When integrating multiple heuristics, **DSAT-Set1** faces a larger action space due to the increased number of tunable options. As solving time increases, the accumulated exploration allows the MAB framework to refine more accurate expected rewards, enabling **DSAT-Set1** to select



■ **Figure 4** Performance comparison between tuning single and multiple heuristics.

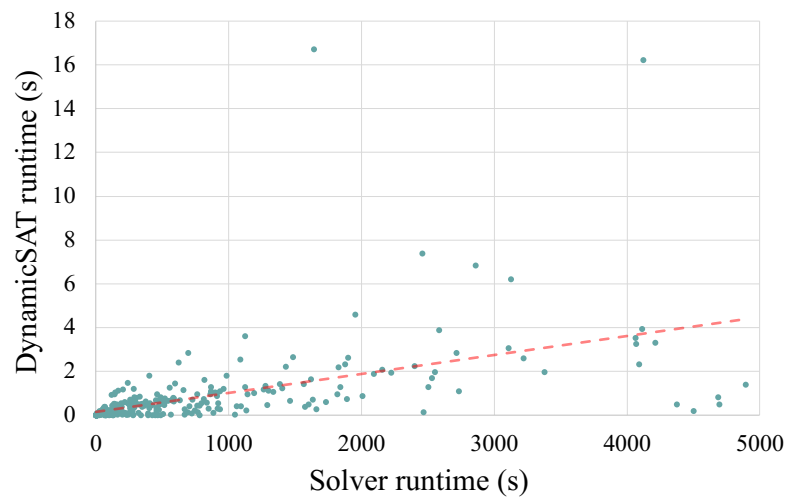
more suitable actions. The same observation can be found in **DSAT-Set2**. Therefore, the experimental results suggest that DynamicSAT could benefit more on hard problem instances given the multiple tunable heuristics.

Due to page limitations, we only investigate the effectiveness and generalization ability of DynamicSAT across various heuristics in this work, leaving the exploration of synergies for future work.

### A.3 Runtime Analysis

In this subsection, we quantify the computational overhead introduced by the MAB-based configuration tuning process relative to the overall SAT solving runtime. To this end, we instrument our solver implementation to record runtime spent in dynamic configuration tuning, including the Sampling, Decision, and Free stages of the MAB agent (see Line 12-32 in Algorithm 1). The experiments are conducted using the **DSAT-compact** variant across the 400 instances in the 2024 SAT Competition Benchmark. The experimental settings remain consistent with those described in previous sections.

Out of the 400 instances, our solver successfully solves 316 cases, which are included in the subsequent runtime analysis. We compare the total runtime of our solver with that of the baseline DynamicSAT system and present the results in Figure 5. The runtime of DynamicSAT exhibits a near-linear relationship with the overall solver runtime, indicating consistent computational behavior across instances. Furthermore, the cumulative time spent specifically on DynamicSAT is only 227.87 seconds, compared to a total solver runtime of 171,222.46 seconds across all 316 solved instances. This accounts for merely 0.13% of the total execution time, indicating that the overhead introduced by the MAB-based tuning mechanism is negligible in practice.



■ **Figure 5** Runtime comparison between solver and DynamicSAT.

#### A.4 Instance Families

See Table 8. We only list the representative families with a significant number of problem instances. The families containing fewer instances and uncategorized instances, are not included in this analysis.

■ **Table 7** Tunable Heuristic Configurations and Parameters.

Heuristic	Configuration	Type	Range	Default	Function
backbone	backbone	Bool	[0,1]	1	Enable backbone analysis
	backboneeffort	Int	[0,1e5]	20	Effort level
	backbonemaxrounds	Int	[1,1e6]	1k	Maximum analysis rounds
	backbonerounds	Int	[1,1e6]	100	Base analysis rounds
bump	bump	Bool	[0,1]	1	Enable clause bump
	bumpreasons	Bool	[0,1]	1	Track bump reasons
	bumpreasonslimit	Int	[1,10]	10	Reason list size limit
	bumpreasonsrate	Int	[1,10]	10	Reason decay rate
chrono	chrono	Bool	[0,1]	1	Chronological backtracking
	chronolevels	Int	[0,1e7]	100	Maximum jumped levels
compact	compact	Bool	[0,1]	1	Compaction strategy
	compactlim	Int	[0,100]	10	Compaction threshold
eliminate	eliminate	Bool	[0,1]	1	Enable eliminate strategy
	eliminatebound	Int	[0,8k]	16	Variable elimination bound
	eliminatecslim	Int	[1,100]	100	Clause size limit
	eliminateeffort	Int	[0,100]	100	Elimination effort
	eliminateinit	Int	[0,500]	500	Initial elimination interval
	eliminateint	Int	[10,500]	500	Stable elimination interval
	eliminateocclim	Int	[0,2k]	2k	Literal occurrence limit
	eliminatorounds	Int	[1,1e3]	2	Maximum elimination rounds
factor	factor	Bool	[0,1]	1	Enable factor strategy
	factorcandrounds	Int	[0,2]	2	Candidate selection rounds
	factoreffort	Int	[0,1e6]	50	Factorization effort
	factorhops	Int	[1,10]	3	Hyper binary resolution hops
	factoriniticks	Int	[700,1e6]	700	Initial conflict ticks
	factorsize	Int	[5,100]	5	Maximum factor size
	factorstructural	Bool	[0,1]	0	Structural factorization
reorder	reorder	Bool	[0,1]	1	Enable reorder strategy
	reorderinit	Int	[0,1e5]	10k	Initial reorder interval
	reorderint	Int	[1,1e5]	10k	Stable reorder interval
	reordermaxsize	Int	[2,256]	100	Maximum cache size
rephase	rephase	Bool	[0,1]	1	Enable rephase strategy
	rephaseinit	Int	[10,1e5]	1k	Initial phase reset interval
	rephaseint	Int	[10,1e5]	1k	Stable phase reset interval
transitive	transitive	Bool	[0,1]	1	Enable transitive strategy
	transitiveeffort	Int	[0,2k]	20	Transitive reduction effort
	transitivekeep	Bool	[0,1]	1	Keep transitive clauses
vivify	vivify	Bool	[0,1]	1	Enable clause vivification
	vivifyeffort	Int	[0,1e3]	100	Vivification effort level
	vivifyirred	Int	[1,100]	1	Irredundant clause effort

**Table 8** Par2 Score comparison of different variants across instance families in 2024 SAT Competition Benchmark.

Family	Count	baseline	DSAT-backbone		DSAT-bump		DSAT-chrono		DSAT-compact		DSAT-eliminate	
			Par2	Red.	Par2	Red.	Par2	Red.	Par2	Red.	Par2	Red.
Cryptography	14	1,310.32	414.24	68.39%	659.13	49.70%	2,058.86	-57.13%	1,145.65	12.57%	1,216.48	7.16%
Scheduling	28	1,183.02	830.43	29.80%	700.52	40.79%	815.77	31.04%	1,056.72	10.68%	814.91	31.12%
Maxsat	13	2,014.63	907.72	54.94%	1,184.61	41.20%	1,162.92	42.28%	717.49	64.39%	1,061.59	47.31%
Software verification	13	609.20	408.20	32.99%	390.84	35.84%	929.61	-52.60%	347.31	42.99%	398.08	34.65%
Random circuits	15	1,243.49	1,029.34	17.22%	1,114.68	10.36%	905.38	27.19%	913.58	26.53%	1,068.19	14.10%
Miter	40	1,010.72	618.64	38.79%	948.85	6.12%	606.90	39.95%	580.50	42.57%	688.30	31.90%
Hamiltonian	41	404.76	243.36	39.88%	546.41	-35.00%	368.63	8.93%	233.82	42.23%	691.99	-70.96%
Argumentation	21	1,657.23	858.34	48.21%	2,359.05	-42.35%	1,013.27	38.86%	831.37	49.83%	1,044.30	36.98%
Independent set	7	2,373.78	1,985.18	16.37%	2,087.52	12.06%	1,788.86	24.64%	1,788.56	24.65%	1,828.35	22.98%

Family	Count	baseline	DSAT-factor		DSAT-reorder		DSAT-rephase		DSAT-transitive		DSAT-vivify	
			Par2	Red.	Par2	Red.	Par2	Red.	Par2	Red.	Par2	Red.
Cryptography	14	1,310.32	563.59	56.99%	1,296.83	1.03%	1,434.81	-9.50%	546.71	58.28%	508.96	61.16%
Scheduling	28	1,183.02	815.79	31.04%	797.84	32.56%	887.24	25.00%	909.80	23.10%	1,045.13	11.66%
Maxsat	13	2,014.63	919.30	54.37%	822.33	59.18%	1,157.78	42.53%	850.46	57.79%	1,310.09	34.97%
Software verification	13	609.20	431.53	29.16%	364.91	40.10%	475.02	22.03%	394.05	35.32%	518.77	14.84%
Random circuits	15	1,243.49	925.03	25.61%	958.47	22.92%	1,003.53	19.30%	886.37	28.72%	2,901.92	-133.37%
Miter	40	1,010.72	637.64	36.91%	576.27	42.98%	729.32	27.84%	591.76	41.45%	1,232.54	-21.95%
Hamiltonian	41	404.76	328.48	18.85%	274.95	32.07%	305.72	24.47%	198.25	51.02%	472.29	-16.68%
Argumentation	21	1,657.23	914.95	44.79%	822.04	50.40%	1,099.66	33.64%	943.72	43.05%	1,524.36	8.02%
Independent set	7	2,373.78	1,931.14	18.65%	1,847.57	22.17%	1,870.54	21.20%	1,869.59	21.24%	2,044.19	13.88%