Multi-armed Bandit Algorithms for the Boolean Satisfiability Problem: A Survey

Zhihui Xie, Xu Liu, Shuai Li*

Shanghai Jiao Tong University
Shanghai, China
{fffffarmer, liu_skywalker, shuaili8}@sjtu.edu.cn

Abstract

This paper provides a survey of recent literature on the use of multi-armed bandit algorithms to solve the Boolean satisfiability problem (SAT), a well-known NP-complete problem with broad applications in academia and industry. The application of bandit algorithms in modern SAT solvers has achieved great success in recent years, as evidenced by the excellent performance of SAT solvers using bandit algorithms in SAT competitions. Bandit algorithms are classic randomized optimization algorithms that strike a balance between exploration and exploitation and can aid in designing and improving heuristics in SAT solvers. In this paper, we introduce several aspects of the application of bandit algorithms in modern SAT solvers, ranging from heuristic methods in CDCL and SLS solvers to strategies in parallel SAT solvers. The use of bandit algorithms in SAT solvers still holds great potential. In conclusion of the survey, we summarize the current issues and suggest possible future research directions.

Introduction

Boolean satisfiability problem (SAT) is one of the most famous NP-complete problems, which was proven in Cook-Levin Theorem in 1971 (Cook 1971). As a powerful tool for solving the SAT instances, SAT solvers have been widely used in academic research, artificial intelligence, and industry. Artificial intelligence planning is one of the earliest fields to apply SAT solvers (Kautz, Selman et al. 1992). In the electronic design automation (EDA) industry, SAT solvers are standard tools for formal equivalence checking and model checking (Marques-Silva and Sakallah 2000). SAT solvers are also used in a broader area of software and hardware verification (D'silva, Kroening, and Weissenbacher 2008; Gupta, Ganai, and Wang 2006) and solving mathematical problems (Zhang 2021).

The basic idea of solving SAT is to explore the search space in non-polynomial complexity since the P versus NP problem remains unsolved. Modern SAT solvers are primarily based on the Conflict-Driven Clause Learning (CDCL) algorithm (Marques-Silva and Sakallah 1999), which is an extension to the Davis-Putnam-Logemann-Loveland (DPLL) algorithm (Davis, Logemann, and Loveland 1962). DPLL is a simple backtracking-based search algorithm. CDCL extends it with clause learning, namely, updating the original SAT problem's formula (i.e. clauses) with newly gained information from an unsatisfiable assignment to accelerate the subsequent search process. CDCL maintains current partial assignments, performs recursive traversing and backtracking, making it powerful for structured formulas. The Stochastic Local Search (SLS) algorithm (Kautz, Sabharwal, and Selman 2009) is also a major approach for SAT solving, which flips the value of assigned variables to find a satisfiable assignment. This approach allows SLS to quickly find possible solutions for unstructured hard formulas. Although CDCL and SLS can solve SAT problems independently (Marques-Silva and Sakallah 1999), modern SAT solvers incorporate both paradigms into the solving process to leverage their respective strengths. Taking the state-of-the-art SAT solver Kissat (Fleury and Heisinger 2020) as an example, modern high-performance SAT solvers employ CDCL in the main solving process and start a strategy referred to as "walking" under certain conditions, during which SLS is employed for a period of exploration.

Compared to solvers decades ago, the performance of SAT solvers has significantly improved in recent years, largely due to the application of better heuristic strategies (Marques-Silva, Lynce, and Malik 2021). Heuristic strategies are widely applied in modern SAT solvers, including branching heuristics, restart policy of CDCL solvers, information sharing between sub-solvers of parallel SAT solvers, etc (Biere, Heule, and van Maaren 2009). Branching heuristics determine the assignment order of unassigned variables during the search process. Restart policies discard current assignments within the search process to avoid getting stuck in local minima and explore different parts of the search space. In parallel SAT solvers, information sharing governs when, where, and what kind of information is shared among sub-solvers to expedite the search processes. These heuristics have been shown to be crucial in modern SAT solvers (Fleury and Heisinger 2020).

In recent years, multi-armed bandit (MAB) algorithms have shown great performance in designing new heuristics and optimizing old heuristics. Bandit algorithms aid in decision-making during the search process by addressing the exploration-exploitation trade-off. The exploration-exploitation trade-off in bandit algorithms involves balanc-

*Corresponding author.

ing the act of exploring the search space to identify regions in which good solutions may be found (exploration) and putting the knowledge gained from exploration to use to maximize rewards (exploitation) (Macready and Wolpert 1998).

Two branching heuristics, the conflict history based (CHB) (Liang et al. 2016a) branching heuristic and the learning rate based (LRB) (Liang et al. 2016b) branching heuristic, are designed using bandit algorithms. In SAT Competition 2021, Kissat_MAB achieved first place in the main track by combining two branching heuristics (VSIDS and CHB) in the restart process of CDCL through bandit algorithms. In the established SAT Competition (Järvisalo et al. 2012), SAT solvers that integrate bandit algorithms have shown highly competitive performance in recent years and continue to grow (see Figure 1).

The success of bandit-based SAT solvers has attracted great attention from researchers. However, we notice that there is no relevant survey on bandit algorithms for SAT solvers in the existing surveys (Belle 2020). Hence our summary focuses on the application of bandit algorithms in SAT solving, including an overview of commonly used bandit algorithms in modern SAT solvers, a summary of SAT solvers built with bandit algorithms, and a discussion of the methods to apply bandit algorithms in SAT solving.

Preliminaries

This section presents the fundamental definitions of SAT and outlines the key techniques used in modern SAT solvers, emphasizing the heuristics that have been or have the potential to be optimized using bandit algorithms. Additionally, a brief introduction to bandit algorithms is provided.

Boolean Satisfiability Problem

Boolean Satisfiability Problem (SAT) is a fundamental problem in computer science, which aims to determine whether a Boolean formula can be satisfied by assigning a Boolean value to each variable that makes the entire formula evaluate to True. Every Boolean formula can be given in the Conjunctive Normal Form (CNF). A CNF formula is a conjunction (AND) of one or more clauses, where a clause is a disjunction (OR) of literals. A literal is a variable or its negation (NOT). For example, one can verify that the given CNF formula $(x \vee y) \wedge (\neg x \vee y)$ has a solution by assigning x to be True or False and y to be True, but the CNF formula $(x \vee y) \wedge (\neg x \vee y) \wedge (\neg y)$ has no solution, since there are no such values of x and y to make the formula true. For further information on the basic concepts of SAT and the technical details of SAT solvers, readers can refer to (Biere, Heule, and van Maaren 2009).

Key Techniques in Modern SAT Solvers

As the P versus NP problem remains unsolved, the NP-complete problem of SAT currently lacks a polynomial-complexity solving algorithm and is mainly solved using search-based methods, which have exponential worst-case complexity. The main idea behind search-based methods is to explore the search space of SAT and optimize the exploration using heuristic methods.

Algorithm 1: The SAT Solving Process†

```
Input: A CNF formula φ
Function CDCL (φ)
    ν ← InitialWalk (φ);
    res ← NULL;
    while res is NULL do
        Assign a selected variable.
        conflict ← UnitPropagation (φ, ν);
        if conflict ≠ ∅ then
            res ← ConflictAnalysis (φ, ν,
              conflict);
        if meeting the conditions of restart then
            Restart (φ, ν);
        if meeting the conditions of rephase then
            // local search is called in
               rephase
            Rephase (φ, ν);
        if all variables are assigned then
            res ← SAT;
    return res;
Function ConflictAnalysis (φ, ν, conflict)
    Determine backtrack level j;
    if conflict at level 0 then
        return UNSAT;
    Find, minimize, and learn a new clause;
    Backtrack to level j;
    Update variable score of branching heuristics;
    return NULL;
```

† taking the simplified version of Kissat (Fleury and Heisinger 2020) as example

CDCL and SLS are the two main paradigms used in modern SAT solvers. CDCL maintains current partial assignments, performs recursive traversing and backtracking, making it powerful for structured formulas. On the other hand, SLS starts with a random full assignment and then flips variable values until satisfying the formula. This approach allows SLS to quickly find possible solutions for unstructured hard formulas. Although they can solve SAT problems independently (Marques-Silva and Sakallah 1999), modern SAT solvers incorporate both paradigms into the solving process to leverage their respective strengths. For instance, in Kissat (Fleury and Heisinger 2020), CDCL is the main solver while a local search solver is called when saved phases are reset by the solver; this local search process is only used once immediately after its execution. Algorithm 1 presents a simplified SAT solving process using Kissat as an example.

Clause Learning Conflict-Driven Clause Learning (CDCL) (Marques-Silva and Sakallah 1999), an extension of Davis-Putnam-Logemann-Loveland (DPLL) (Davis, Logemann, and Loveland 1962), is a cornerstone of SAT solving. While DPLL systematically explores the search

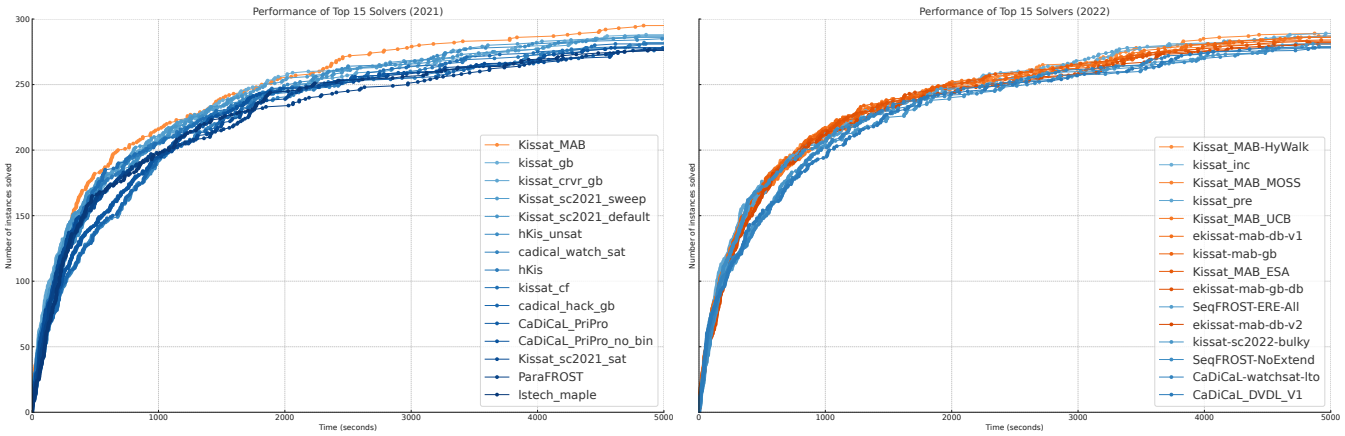

Figure 1: Top 15 Solvers in SAT Competition 2021 and 2022. Solvers are ranked by the number of instances solved. Solvers using bandit algorithms are marked with warm colors, while other solvers are marked with cool colors. The more instances are solved within a given runtime limit, the better the solver is.

space for a satisfying assignment of a given CNF formula, CDCL enhances it by learning from conflicts. When a conflict arises (i.e., a clause cannot be satisfied under the current assignment), CDCL adds a new learned clause to the formula, preventing the algorithm from revisiting invalid search paths and accelerating convergence. CDCL constructs an implication graph during the search process to represent relationships between variables. Vertices correspond to assigned variables, and edges indicate how variable assignments depend on others. For example, given the clause $x \vee \neg y \vee z$, if $x = \mathsf{False}$ and $y = \mathsf{True}$, it follows that $z = \mathsf{True}$, with edges drawn from x and y to z. When a conflict occurs, represented by edges leading to a conflict node κ, the algorithm backtracks and identifies the variables responsible. Their negations form a new learned clause, which is added to the formula. This process, known as clause learning, corresponds to identifying a cut in the implication graph. CDCL often uses strategies like the 1st-UIP (Unique Implication Point) rule (Möhle and Biere 2019) to determine backtracking points and which clause to learn, optimizing both efficiency and performance.

Variable and Phase Selection Heuristics The branching heuristics, also known as variable selection heuristics, aim to select the most "appropriate" variable during the search process to accelerate the exploration of the search space. One common type of the branching heuristic is to maintain a score for each variable and select the unassigned variable with the highest score. The VSIDS (Moskewicz et al. 2001) heuristic, as well as the bandit-based CHB and LRB heuristics, are the most widely used branching heuristics. Since clause learning is vital in CDCL algorithms, all three heuristics determine variable scores based on the efficiency of clause learning.

Phase selection heuristics aim to determine the Boolean values (True or False) of variables generated by variable selection. The speed of the search can be influenced by assigning different values. Moreover, the value of variables that have been used before can be reused, which is known as phase saving (Pipatsrisawat and Darwiche 2007). The concept of rephasing was proposed in (Biere 2017), which uses multiple strategies (initial, inverted initial, flipped, random, best phases, and local search) to replace the conflict intervals of saved phases.

Restart Policy CDCL solvers implement a strategy called "restart" in which the current assignment is discarded while other states, such as learned clauses and saved phases, are preserved. Restarting the search helps SAT solvers overcome local minima and complex search spaces by exploring different parts of the search space. Different search paths may have different search speeds. The use of restart in SAT solvers has achieved significant success in practice, thus modern SAT solvers commonly employ restart strategies such as Luby restart (Luby, Sinclair, and Zuckerman 1993) and Glucose restart (Audemard and Simon 2012) to determine the intervals of restarts.

Clause Deletion Clause deletion is a crucial aspect of CDCL solvers since an excessive number of clauses results in low efficiency in Boolean Constraint Propagation (BCP) (Krüger, Lorenz, and Wörz 2022). In addition to simple techniques such as deleting clauses that are supersets of other clauses, clause deletion also relies on assessing the quality of clauses. One metric used for this purpose is Literal Block Distance (LBD) (Audemard and Simon 2009), which is defined as the number of decision levels of the clause. The concept of LBD is that decision levels regularly decrease during the search. Therefore, clauses that appear at lower decision levels are more likely to be activated in BCP. Specifically, clauses with an LBD of 2 are considered to be of high quality and are referred to as glue clauses. These clauses are not typically removed during clause deletion.

Stochastic Local Search Stochastic Local Search (SLS) algorithms are incomplete, meaning they do not guarantee finding a satisfying assignment or proving unsatisfiability. Typically, they are biased toward satisfiability: run with a resource limit, they either return a solution or fail,

Year	# Solvers	Best Ranking
2021	1	1/48
2022	9	1/50
2023	9	2/49

Table 1: Statistics of bandit-based SAT solvers in SAT Competition 2021, 2022, and 2023.

but never declare the formula unsatisfiable (Kautz, Sabharwal, and Selman 2009). SLS flips variable assignments to find a satisfying solution, starting with a random or given assignment. Variables are selected to flip either randomly (random walk) or based on maximizing scores (greedy), calculated by the number of satisfied/unsatisfied clauses. Like CDCL solvers, SLS algorithms rely on variable selection and restart policies. In modern SAT solvers, SLS often complements CDCL, e.g., by periodically using SLS with the current CDCL assignment, as implemented in solvers like Kissat.

SAT as Sequential Decision-Making Problems

Sequential decision-making involves determining a series of actions, where each decision affects future outcomes, with both immediate and long-term consequences. Many strategies in SAT solving can be viewed as sequential decision problems. Key examples include variable decision (branching heuristics), which selects variables to assign during the search to optimize future efficiency; selecting branching heuristics during restarts to navigate the search space effectively; and other strategies such as rephasing heuristics, restart policies, and variable flipping in local search.

The following sections introduce the bandit algorithm, a powerful tool for addressing sequential decision problems, and survey several bandit-based SAT solvers. These solvers apply bandit algorithms to optimize the aforementioned strategies and have demonstrated significant success in SAT competitions. Table 1 summarizes the performance of bandit-based SAT solvers in recent competitions.

Bandit Algorithms

The multi-armed bandit (MAB) is a widely used framework for addressing the exploration-exploitation trade-off under uncertainty. Consider a scenario where you face two slot machines (a two-armed bandit), and each arm provides a random reward based on its probability distribution each time it is pulled. The problem of determining which arm to pull next to maximize your reward while repeatedly pulling arms is referred to as the bandit problem. Therefore, the bandit algorithm needs to address the exploration-exploitation trade-off, that is, decide whether to exploit the best arm based on historical rewards or explore other arms that may yield higher rewards in the future, in order to strike a balance between exploration and exploitation.

To evaluate the quality of a bandit algorithm, we define the regret of a policy as the difference between the expected reward obtained by the best arm and the expected actual reward obtained by the algorithm's decisions. The regret of policy π on bandit instance ν is given by:

$$R_n(\pi, \nu) = n\mu^\star(\nu) - \mathbb{E}\left[\sum_{t=1}^{n} X_t\right],$$

where $\mu^\star$ denotes the largest mean of all arms and X_t denotes the reward obtained in round t.

The Upper Confidence Bound (UCB) algorithm is one of the most common and effective bandit algorithms used in SAT solvers. UCB1 is a widely used version of UCB in SAT solvers. The core idea of UCB is to calculate a score for each arm and pull the arm with the highest score. Let $\widehat{\mu}_i(t)$ be the average of rewards of arm i, $i = 1, 2, \ldots, K$, in the first t rounds. The algorithm starts by playing each arm once. Then, the score $\mathrm{UCB1}$ is defined as:

$$\mathrm{UCB1}_i(t) = \widehat{\mu}_i(t) + \sqrt{\frac{4\ln(t)}{T_i(t)}},$$

where $T_i(t)$ denotes the number of times the i-th arm has been pulled up to round t. In round t, select arm $\mathrm{argmax}_i\, \mathrm{UCB1}_i(t-1)$ and the reward is X_t.

Algorithm 2: UCB1†

Input: k
Choose each arm once.
for $t = 1, 2, \ldots$ **do**
  Compute $A_t = \mathrm{argmax}_i\, \mathrm{UCB1}_i(t-1)$;
  Observe reward X_t from arm A_t;
  Update upper confidence bounds;

† adapted from (Lattimore and Szepesvári 2020)

Another widely used UCB strategy MOSS (Minimax Optimal Strategy in the Stochastic Case) takes into account the number of arms K. After choosing each arm once, MOSS applies the following score:

$$\mathrm{MOSS}_i(t) = \widehat{\mu}_i(t) + \sqrt{\frac{4\ln(\max\{\frac{t}{KT_i(t)}, 1\})}{T_i(t)}}.$$

It is proved in (Auer, Cesa-Bianchi, and Fischer 2002) that if policy UCB1 is run on K machines having arbitrary reward distributions $P_1, \ldots, P_k$ with support in $[0, 1]$, then its expected regret after any number n of plays has an expected cumulative regret no worse than $O(\sqrt{KT\ln T})$. Regret analysis reveals that UCB1 demonstrates great performance. MOSS guarantees a better expected cumulative regret $O(\sqrt{KT})$, and experimental results indicate that MOSS is a robust strategy that improves the performance on each individual benchmark.

The Exponential Recency Weighted Average (ERWA) algorithm, although not typically considered a bandit algorithm, is widely used in SAT solvers due to its simple computational approach and remarkable effectiveness. Let $\mathrm{ERWA}_i(t)$ denote the exponentially decaying weighted average of rewards, where i represents the arm and t denotes

the round. We define the weight $w_i \coloneqq \alpha(1-\alpha)^{n-i}$, and it can be proven that $\sum_{i=1}^{n} w_i = 1$, making w_i well-defined.

$$\mathrm{ERWA}_i(t) \coloneqq \sum_{1 \le j \le i, A_j = i} w_i X_i,$$

where A_j denotes the selected arm in round j.

It is worth noting that the above expression can be simplified. Suppose $A_t = i$ and we wish to calculate $\mathrm{ERWA}_i(t)$. Let t' denote the maximum previous round where the selected arm was i, i.e. $t' \coloneqq \max_{x<t, A_x = i} x$. Then, we can iteratively calculate $\mathrm{ERWA}_i(t)$ as follows:

$$\mathrm{ERWA}_i(t) = (1-\alpha)\mathrm{ERWA}_i(t') + \alpha X_t.$$

In essence, this is simply using a weighted average of the new reward to update the score.

In addition to UCB and ERWA, there are several other efficient bandit algorithms, which can be found in (Slivkins et al. 2019) and (Lattimore and Szepesvári 2020). Further research into selecting bandit algorithms for SAT solvers could be explored in the future.

Bandit Algorithms in SAT Solvers

Bandit algorithms attempt to address the exploration-exploitation trade-off, which perfectly matches the aim of the search process in SAT solvers, i.e. explore the structure of the search space in an efficient way by selecting and optimizing the order of searching to find the optimal search path in shorter time. Bandit algorithms play a role in selecting clauses and variables. Moreover, bandit algorithms assist in selecting among different strategies and parameters during the solving process to combine their benefits. This section reviews some well-designed and efficient SAT solvers using bandit algorithms in recent years and discusses the bandit components in modern SAT solvers.

Bandit in Branching Heuristics

The key to variable selection lies in the design of the variable scoring calculation. Variable State Independent Decaying Sum (VSIDS) (Moskewicz et al. 2001) is one of the most widely used branching heuristics. Its basic idea is to calculate a score for each variable, initially set to 0, and increase it by 1 when it is bumped. Different SAT solvers bump different variables; some bump variables in the learned clause, while others bump variables in the learned clause and variables that are resolved in determining the learned clause. After each conflict, VSIDS also decays the score of all variables by multiplying them with a decay factor to increase the locality of the scores.

However, VSIDS does not fully utilize the historical information. The periodic decay in VSIDS causes the impact of historical conflicts on the current search process to decrease rapidly over time. Meanwhile, VSIDS merely focuses on the number of conflicts a variable is involved in, without considering the efficiency of conflict generation for a variable. Branching heuristics that have a higher rate of conflict clause generation per unit time are more effective than ones that have a lower rate (Liang et al. 2016a). Therefore, two branching heuristics, conflict history-based branching heuristic (CHB) (Liang et al. 2016a) and learning rate based branching heuristic (LRB) (Liang et al. 2016b) have been designed.

Based on conflict history and taking into account the rate of clause learning, the bandit-based heuristics CHB and LRB are more adaptive than VSIDS. Despite specific definition differences, the reward r_v for a variable v (i.e. the reward of an arm in bandit) is defined as the rate of learning, a metric of the impact of v on the speed of clause learning in both CHB and LRB.

CHB updates r_v and subsequently the score when a variable is updated (branched on, propagated, or asserted). LRB updates r_v and score when a variable is unassigned. The score of variable Q_v is defined by ERWA, as

$$Q_v = (1-\alpha)Q_v + \alpha r_v, \alpha = 0.4$$

Experimental results show that LRB performs better than CHB, and both heuristics solve more instances than VSIDS within the time limit (Liang et al. 2016b,a). Some experimental results (Cherif, Habet, and Terrioux 2021) suggest that CHB is more robust with respect to different solvers and settings. It is also shown that different branching heuristics exhibit significant differences in performance on different instance families (Cherif, Habet, and Terrioux 2021).

Bandit in Selecting Branching Heuristics

Since a heuristic often performs well on a certain type of instance while behaving poorly on another, as is described above, solvers that combine different heuristics such as MapleCOMSPS (Liang et al. 2016c) have shown promising results. However, early attempts were based on random and static strategies. For example, the random strategies select a branching heuristic at each restart randomly, while single switch strategies switch the heuristic after a period of time (Cherif, Habet, and Terrioux 2021).

Kissat_MAB (Cherif, Habet, and Terrioux 2021) employs UCB in restart to switch branching heuristic between VSIDS and CHB for the next search to take the advantage of both heuristics, picking an adequate heuristic among CHB and VSIDS for each instance. Because assignments are cleared at restart, the solver can use a new heuristic to begin exploring the search space in different search paths. In terms of implementation details, Kissat_MAB maintains scores of variables for VSIDS and CHB respectively, which are not cleared at restart. The two heuristics use their own scores to select the next variable to assign.

Kissat_MAB uses a two-armed bandit framework to select the next heuristic. Let a and t denote the current heuristic and the current run, $a = 1, 2, t = 1, 2, \ldots$, then the reward X_t is defined as

$$X_t = \frac{\log_2(decisions_t)}{decidedVars_t},$$

where $decisions_t$ and $decidedVars_t$ respectively denote the number of decisions in the current run and the number of variables that have been selected. The concept of r, calculated by $decisions_t$ and $decidedVars_t$, is well-defined, since $decisions_t \le 2^{decidedVars_t}$.

Methods or Solvers	Bandit Algorithms	Bandit Components	Fundamental Algorithms
CHB (Liang et al. 2016a)	ERWA	Branching Heuristic	CDCL
LRB (Liang et al. 2016b)	ERWA	Branching Heuristic	CDCL
Kissat_MAB (Cherif, Habet, and Terrioux 2021)	UCB	Selecting Branching Heuristic	CDCL
MapleSSV (Nejati, Chowdhury, and Ganesh 2021)	UCB	Restart Policy	CDCL
Kissat-Adaptive-Restart (Li et al. 2022)	UCB	Restart Policy	CDCL
Kissat-MAB-rephasing (Chen et al. 2022)	UCB	Rephasing Heuristic	CDCL
BandSAT (Zheng et al. 2022a)	UCB	Variable Selection	SLS

Table 2: Summary of reviewed SAT solvers built with multi-armed bandit techniques.

The use of bandit in adaptive selecting branching heuristics at restart has been shown to be effective. The combination of VSIDS and CHB has also been shown to be compatible. In SAT competition 2021, Kissat_MAB achieved first place with UCB1 in the main track. Furthermore, Kissat_MAB with MOSS outperforms UCB. In SAT competition 2022, Kissat_MAB_MOSS and Kissat_MAB_UCB (using UCB1) achieved the 5th and 6th places respectively, second only to solvers that optimize other heuristics based on Kissat_MAB. The idea in Kissat_MAB that establishes appropriate metrics for stages of the SAT solving process in order to evaluate its efficiency, and then applies the bandit framework to make selections from the scores calculated from the metric, also provides inspiration for future research.

Bandit in Rephasing Heuristics

Rephasing (Biere 2017) replaces saved phases in the phase-saving mechanism during conflict intervals. In the high-performance SAT solver Kissat, rephasing follows a fixed, handcrafted order: $(\texttt{BWOBWIBW\#BWF})^{\omega}$ (Biere and Fleury 2020). $\texttt{B}, \texttt{W}, \texttt{O}, \texttt{I}, \#, \texttt{F}$ respectively refer to the best phases (replacing saved phasing with cached best assignment found so far), local search (implementing the ProbSAT strategy (Balint 2014); phase W standing for walk), original value (setting all saved phases to the original value, i.e. True on default in Kissat), inverted value (setting all saved phases to the opposite value, i.e. False on default in Kissat), randomizing values (randomizing the saved phase), flipping values (flipping the saved values).

Building on Kissat_MAB's success, Kissat-MAB-rephasing (Chen et al. 2022) employs a four-armed bandit to adaptively select rephasing strategies among $\texttt{B}, \texttt{I}, \texttt{O}, \texttt{W}$. Unlike Kissat_MAB, which selects branching heuristics during restarts, Kissat-MAB-rephasing focuses solely on rephasing heuristic selection. This selection, guided by UCB, occurs independently of restarts, using modified metrics ($decisions_t$ and $decidedVars_t$) to evaluate solver performance during rephasing. These metrics are reset during each rephasing period to ensure accurate assessment.

Bandit in Restart Policy

Restart policies dictate when and how often to restart SAT solvers. Common strategies include geometric restarts, which increase restart intervals geometrically, and Luby restarts (Luby, Sinclair, and Zuckerman 1993), which follow the Luby sequence. EMA restarts (Biere and Fröhlich 2015) use the exponential moving average of clause lengths, while Glucose restarts (Audemard and Simon 2012) rely on the literal block distance (LBD) metric, comparing short- and long-term averages. Machine learning-based restarts (MLR) (Liang et al. 2018) predict the quality of upcoming LBDs and restart when predicted quality is low.

Kissat-adaptive-restart (Li et al. 2022) selects restart policies dynamically using UCB1. In Kissat, solvers alternate between stable (Luby restarts) and unstable (Glucose restarts) modes. Kissat-adaptive-restart refines this by letting UCB1 choose between modes during restarts, based on the same metrics(X_t and scores) used in Kissat_MAB. Notably, it does not apply bandit algorithms to branching heuristic selection. MapleSSV (Nejati, Chowdhury, and Ganesh 2021) extends this idea, using UCB to choose among uniform, linear, Luby, and geometric restarts. The reward is based on conflicts and LBD during each restart period:

$$X_t = \frac{\Delta conflicts_t}{\overline{LBD_t}},$$

where $\Delta conflicts_t, \overline{LBD_t}$ are respectively defined as the conflicts encountered and the average LBD in the restart period t.

MapleSSV selects the restart policy for the next run during restarts and then performs the search until the chosen restart policy decides to restart. It is worth noting that there are differences in the constants used in UCB1 between MapleSSV and Kissat_MAB, replacing 4 with 0.4:

$$\mathrm{UCB1}_i(t) = \widehat{\mu}_i(t) + \sqrt{\frac{0.4 \ln t}{T_i(t)}}.$$

The calculation of $\widehat{\mu}$ in MapleSSV also differs from that in Kissat_MAB. In MapleSSV, $\widehat{\mu}$ is defined as follows, with a different definition of μ and T, both decaying by a factor of $\alpha = 0.95$:

$$\begin{aligned}
\widehat{\mu}_i(t) &= \frac{\mu_i(t)}{T_i(t)} \\
\mu_i(t) &= \alpha \mu_i(t-1) + X_t \mathbb{I}\{A_t = i\} \\
T_i(t) &= \alpha T_i(t-1) + \mathbb{I}\{A_t = i\},
\end{aligned}$$

where $\mathbb{I}\{A_t = i\}$ is the indicator function.

Bandit in Local Search

One of the most important issues in local search is the strategy of encountering a local optimum. BandMaxSAT (Zheng et al. 2022b) is a solver for MaxSAT problems that uses bandit in variable selection in local search, which also has a version of solving SAT problems called BandSAT (Zheng et al. 2022a).

The local search algorithm based on the variable flipping mechanism selects a variable v and flips its Boolean value. The variable selection procedure can be partitioned into two stages. In the case where the current assignment has not yet reached a local optimum, i.e. when the set $GoodVars \neq \emptyset$, the algorithm chooses v from $GoodVars$. It should be noted that varying algorithms may define $GoodVars$ differently. BandSAT uses a single scoring function $score(x)$ to represent the increase of the total dynamic weight of the satisfied clauses caused by flipping x, and define the set $GoodVars = \{x : score(x) > 0\}$.

Conversely, upon reaching a local optimum, the algorithm selects a falsified clause c and then chooses a variable v from it. After a clause c is determined, BandSAT, along with several other algorithms, select v greedily according to their $score$. Previous algorithms, such as CCAnr (Cai, Luo, and Su 2015), employed the simple random walk strategy to select the clauses to be satisfied. BandSAT replaces the random clause sampling by selecting a clause using bandit algorithms.

It is worth noting that in BandMaxSAT, clauses are divided into soft and hard clauses, and some definitions differ accordingly. However, in BandSAT, no such division exists. BandSAT treats clauses as arms. For a given assignment set $\mathcal{A}$, BandSAT maintains the number of falsified clauses $cost(\mathcal{A})$. The reward X_t is calculated as follows:

$$X_t = \frac{cost(\mathcal{A}_t) - cost(\mathcal{A}_{t-1})}{cost(\mathcal{A}_{t-1}) + 1},$$

where $\mathcal{A}_t$ denotes the assignment at the end of round t.

BandSAT employs the delayed reward method (Arya and Yang 2020) to calculate $\widehat{\mu_a}(t)$. The delayed reward method updates the estimated value of the last d (20 by default) pulled arms once a reward is obtained.

$$\widehat{\mu_{a_i}}(t) = \widehat{\mu_{a_i}}(t-1) + \gamma^{d-i} X_t, i \in \{1, 2, \ldots, d\},$$

where $\gamma = 0.9$ denotes the reward discount factor. $\{a_1, a_2, \ldots, a_d\}$ are the recently pulled arms, in which a_d is the most recent one.

BandSAT randomly samples d (20 by default) clauses from falsified clauses and subsequently employs UCB1 to select an arm to pull. BandSAT modifies the constant in UCB1:

$$\mathrm{UCB1}_i(t) = \widehat{\mu_i}(t) + 0.1\sqrt{\frac{\ln t}{T_i(t)}}.$$

Bandit in Algorithm Selection

Due to the sensitivity of SAT instances, there is no one-size-fits-all solution for SAT solving. Therefore, algorithm selection (Rice 1976; Xu et al. 2008) becomes particularly important in the context of SAT. Algorithm selection considers the design of meta-algorithms, constructing algorithms to solve a specific instance from a pool or range of potential algorithms. For example, per-instance algorithm selection involves automatically selecting from a set of different algorithms (i.e., a "portfolio") each time a new instance is presented.

Previous work has framed the problem of algorithm selection as a MAB problem in which each algorithm represents an arm and rewards are associated with solving times (Gagliolo and Schmidhuber 2010). Taking a broader perspective, algorithm selection is closely connected to the wider field of AutoML, which aims to automate the entire process of applying machine learning to real-world problems (Feurer et al. 2015). As bandit-based methods (Li et al. 2017) play a key role in AutoML, there is potential for further exploration of these methods in the context of SAT. For instance, rising bandits (Li et al. 2020), rested bandits (Cella, Pontil, and Gentile 2021), and contextual bandits (Foster, Krishnamurthy, and Luo 2019) have shown promise in this area.

Open Questions

Combining Information from Multiple Heuristics

Note that although using the bandit framework to switch heuristics can solve more instances within the time limit, some instances that can be solved using a single heuristic may not be covered after switching. The existing solver Kissat_MAB selects branching heuristic at restart, which only modifies the calculation of the score of variables while leaving other information unaltered. In parallel SAT solvers, a method for sharing information among solvers in different threads, known as clause exchanging (Audemard et al. 2012), is used. An analogous approach may be applied to the solvers that select heuristics using bandit, in lieu of the existing method whereby newly learned clauses are directly appended to the existing set of clauses.

Adversarial Bandit Algorithms in SAT Solvers

Another problem is that the branching heuristics CHB and LRB use only ERWA, a simple algorithm that is not considered a typical bandit algorithm, which does not have a well-established regret analysis and may perform moderately compared to other bandit algorithms. The well-known bandit algorithms UCB1 and MOSS used in Kissat_MAB have demonstrated excellent performance in stochastic bandits. However, the rewards associated with variables in branching heuristics, i.e. the efficiency of learning, are not fixed as they are influenced by the particular assignments chosen and they may change greatly due to the clause learning process.

Therefore, algorithms for adversarial bandit problems such as Exp3 (Lattimore and Szepesvári 2020) may yield superior results. Unlike stochastic bandit frameworks, where the rewards are assumed to be generated from a stationary distribution, in adversarial bandits, the rewards can change unpredictably over time.

Applying Exp3 to branching heuristics in SAT solvers may have great effects. However, the Exp3 algorithm calculates the sampling distribution P_{ti} for arm i in round t, and samples $A_t \sim P_t$. Therefore, unlike UCB, which can be implemented simply using a heap to select the arm with the

maximum score, implementing Exp3 in SAT solvers may require additional techniques, such as more efficient data structures.

Bandit Algorithms in Parallel SAT Solvers

Moreover, there is a lack of research on applying bandit algorithms in parallel SAT solvers. The existing work, BESS (Lazaar et al. 2012), has proposed a mechanism for controlling the exchange of learned clauses among processing units in portfolio-based approaches. BESS uses bandit algorithms to control cooperation topology (pairs of units able to exchange clauses), which has resulted in performance gains. However, modern SAT solvers with high efficiency have not employed bandit-based methods in controlling cooperation topology. Bandit algorithms have shown great potential in SAT solvers as well as other fields, indicating that more work could be done to fully explore their capabilities in parallel SAT solvers.

Conclusion and Future Work

Bandit algorithms have found extensive applications in SAT solvers, optimizing heuristics like branching, phasing, restart policies in CDCL solvers, and variable selection in local search. These advancements have driven notable successes in SAT competitions and industrial use cases, underscoring the potential of bandit-based approaches in SAT solving.

Despite these achievements, challenges persist. The theoretical underpinnings of CDCL heuristics and their impact on solver performance remain poorly understood, partly due to their complexity and dependence on the solver's execution history (Biere, Heule, and van Maaren 2009). Future work could focus on practical performance metrics and heuristic analysis tailored to specific CNF formula types.

Expanding the use of bandit algorithms to additional heuristics presents a promising direction. Potential avenues include refining multi-heuristic strategies, exploring adversarial bandit algorithms, and integrating bandit methods into tasks like clause deletion and variable elimination. Bandit-based dynamic parameter tuning, such as backtracking level selection in CDCL solvers, offers another exciting opportunity. Furthermore, applying bandit algorithms in parallel SAT solvers remains largely unexplored and could unlock further performance gains.

References

Arya, S.; and Yang, Y. 2020. Randomized allocation with nonparametric estimation for contextual multi-armed bandits with delayed rewards. *Statistics & Probability Letters*, 164: 108818.

Audemard, G.; Hoessen, B.; Jabbour, S.; Lagniez, J.-M.; and Piette, C. 2012. Revisiting clause exchange in parallel SAT solving. In *Theory and Applications of Satisfiability Testing–SAT 2012: 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings 15*, 200–213. Springer.

Audemard, G.; and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *Twenty-first international joint conference on artificial intelligence*. Citeseer.

Audemard, G.; and Simon, L. 2012. Refining restarts strategies for SAT and UNSAT. In *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, 118–126. Springer.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47: 235–256.

Balint, A. 2014. *Engineering stochastic local search for the satisfiability problem*. Ph.D. thesis, Universität Ulm.

Belle, V. 2020. Symbolic logic meets machine learning: A brief survey in infinite domains. In *Scalable Uncertainty Management: 14th International Conference, SUM 2020, Bozen-Bolzano, Italy, September 23–25, 2020, Proceedings 14*, 3–16. Springer.

Biere, A. 2017. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018. *Proceedings of SAT Competition*, 14: 316–336.

Biere, A.; and Fleury, M. 2020. Chasing target phases. In *Workshop on the Pragmatics of SAT*.

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

Biere, A.; Heule, M.; and van Maaren, H. 2009. *Handbook of satisfiability*, volume 185. IOS press.

Cai, S.; Luo, C.; and Su, K. 2015. CCAnr: A configuration checking based local search solver for non-random satisfiability. In *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*, 1–8. Springer.

Cella, L.; Pontil, M.; and Gentile, C. 2021. Best model identification: A rested bandit formulation. In *International Conference on Machine Learning*, 1362–1372. PMLR.

Chen, X.; Guo, W.; Luo, W.; Zhen, H.-L.; Li, X.; Yuan, M.; and Yan, J. 2022. Kissat-MAB-rephasing and Kissat relaxed. *SAT COMPETITION 2022*, 35.

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

Cook, S. A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, 151–158.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Communications of the ACM*, 5(7): 394–397.

D'silva, V.; Kroening, D.; and Weissenbacher, G. 2008. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7): 1165–1178.

Feurer, M.; Klein, A.; Eggensperger, K.; Springenberg, J.; Blum, M.; and Hutter, F. 2015. Efficient and robust automated machine learning. *Advances in neural information processing systems*, 28.

Fleury, A. B. K. F. M.; and Heisinger, M. 2020. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. *SAT COMPETITION*, 2020: 50.

Foster, D. J.; Krishnamurthy, A.; and Luo, H. 2019. Model selection for contextual bandits. *Advances in Neural Information Processing Systems*, 32.

Gagliolo, M.; and Schmidhuber, J. 2010. Algorithm selection as a bandit problem with unbounded losses. In *Learning and Intelligent Optimization: 4th International Conference, LION 4, Venice, Italy, January 18-22, 2010. Selected Papers 4*, 82–96. Springer.

Gupta, A.; Ganai, M. K.; and Wang, C. 2006. SAT-based verification methods and applications in hardware verification. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, 108–143. Springer.

Järvisalo, M.; Le Berre, D.; Roussel, O.; and Simon, L. 2012. The international SAT solver competitions. *Ai Magazine*, 33(1): 89–92.

Kautz, H. A.; Sabharwal, A.; and Selman, B. 2009. Incomplete Algorithms. *Handbook of satisfiability*, 185: 185–203.

Kautz, H. A.; Selman, B.; et al. 1992. Planning as Satisfiability. In *ECAI*, volume 92, 359–363. Citeseer.

Krüger, T.; Lorenz, J.-H.; and Wörz, F. 2022. Too much information: Why CDCL solvers need to forget learned clauses. *Plos one*, 17(8): e0272967.

Lattimore, T.; and Szepesvári, C. 2020. *Bandit algorithms*. Cambridge University Press.

Lazaar, N.; Hamadi, Y.; Jabbour, S.; and Sebag, M. 2012. *Cooperation control in parallel SAT solving: a multi-armed bandit approach*. Ph.D. thesis, INRIA.

Li, L.; Jamieson, K.; DeSalvo, G.; Rostamizadeh, A.; and Talwalkar, A. 2017. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1): 6765–6816.

Li, Y.; Jia, Y.; Luo, W.; Zhen, H.-L.; Li, X.; Yuan, M.; and Yan, J. 2022. Kissat Adaptive Restart, Kissat Cfexp: Adaptive Restart Policy and Variable Scoring Improvement. *SAT COMPETITION 2022*, 39.

Li, Y.; Jiang, J.; Gao, J.; Shao, Y.; Zhang, C.; and Cui, B. 2020. Efficient automatic CASH via rising bandits. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 4763–4771.

Liang, J.; Ganesh, V.; Poupart, P.; and Czarnecki, K. 2016a. Exponential recency weighted average branching heuristic for SAT solvers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.

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

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

Liang, J. H.; Oh, C.; Mathew, M.; Thomas, C.; Li, C.; and Ganesh, V. 2018. Machine learning-based restart policy for CDCL SAT solvers. 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*, 94–110. Springer.

Luby, M.; Sinclair, A.; and Zuckerman, D. 1993. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4): 173–180.

Macready, W. G.; and Wolpert, D. H. 1998. Bandit problems and the exploration/exploitation tradeoff. *IEEE Transactions on evolutionary computation*, 2(1): 2–22.

Marques-Silva, J.; Lynce, I.; and Malik, S. 2021. Conflict-driven clause learning SAT solvers. In *Handbook of satisfiability*, 133–182. IOS press.

Marques-Silva, J. P.; and Sakallah, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5): 506–521.

Marques-Silva, J. P.; and Sakallah, K. A. 2000. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Annual Design Automation Conference*, 675–680.

Möhle, S.; and Biere, A. 2019. Backing backtracking. In *Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22*, 250–266. Springer.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, 530–535.

Nejati, S.; Chowdhury, M. S.; and Ganesh, V. 2021. MapleSSV SAT Solver for SAT Competition 2021. *SAT COMPETITION 2021*, 35.

Pipatsrisawat, K.; and Darwiche, A. 2007. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing–SAT 2007: 10th International Conference, Lisbon, Portugal, May 28-31, 2007. Proceedings 10*, 294–299. Springer.

Rice, J. R. 1976. The algorithm selection problem. In *Advances in computers*, volume 15, 65–118. Elsevier.

Slivkins, A.; et al. 2019. Introduction to multi-armed bandits. *Foundations and Trends® in Machine Learning*, 12(1-2): 1–286.

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2008. SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research*, 32: 565–606.

Zhang, H. 2021. Combinatorial designs by sat solvers 1. In *Handbook of Satisfiability*, 819–858. IOS Press.

Zheng, J.; He, K.; Chen, Z.; Zhou, J.; and Li, C.-M. 2022a. Combining Hybrid Walking Strategy with Kissat MAB, CaDiCaL, and LStech-Maple. *SAT COMPETITION 2022*, 20.

Zheng, J.; He, K.; Zhou, J.; Jin, Y.; Li, C.-m.; and Manya, F. 2022b. Bandmaxsat: A local search MaxSAT solver with multi-armed bandit. *arXiv preprint arXiv:2201.05544*.