Journal of Artificial Intelligence Research 76 (2023) 435-470 Submitted 05/2022; published 01/2023

Generating Random SAT Instances: Multiple Solutions could be Predefined and Deeply Hidden

Dongdong Zhao ZDD@WHUT.EDU.CN
1. Hubei Key Laboratory of Transportation Internet of Things
School of Computer Science and Technology
Wuhan University of Technology
2. University of Science and Technology of China

Lei Liao LIAOLEI666@WHUT.EDU.CN
School of Computer Science and Technology
Wuhan University of Technology

Wenjian Luo LUOWENJIAN@HIT.EDU.CN
(Corresponding author)
School of Computer Science and Technology
Harbin Institute of Technology, Shenzhen

Jianwen Xiang JWXIANG@WHUT.EDU.CN
Hubei Key Laboratory of Transportation Internet of Things
School of Computer Science and Technology
Wuhan University of Technology

Hao Jiang HAOJIANG@AHU.EDU.CN
Key Laboratory of Intelligent Computing and Signal Processing of Ministry of Education, School of Artificial Intelligence, Anhui University

Xiaoyi Hu HUXIAOYI@WHUT.EDU.CN
School of Computer Science and Technology
Wuhan University of Technology

Abstract

The generation of SAT instances is an important issue in computer science, and it is useful for researchers to verify the effectiveness of SAT solvers. Addressing this issue could inspire researchers to propose new search strategies. SAT problems exist in various real-world applications, some of which have more than one solution. However, although several algorithms for generating random SAT instances have been proposed, few can be used to generate hard instances that have multiple predefined solutions. In this paper, we propose the KHidden-M algorithm to generate SAT instances with multiple predefined solutions that could be hard to solve by the local search strategy when the number of predefined solutions is small enough and the Hamming distance between them is not less than half of the solution length. Specifically, first, we generate an SAT instance that is satisfied by all of the predefined solutions. Next, if the generated SAT instance does

not satisfy the hardness condition, then a strategy will be conducted to adjust clauses through multiple iterations to improve the hardness of the whole instance. We propose three strategies to generate the SAT instance in the first part. The first strategy is called the random strategy, which randomly generates clauses that are satisfied by all of the predefined solutions. The other two strategies are called the estimating strategy and greedy strategy, and using them, we attempt to generate an instance that directly satisfies or is closer to the hardness condition for the local search strategy. We employ two SAT solvers (i.e., WalkSAT and Kissat) to investigate the hardness of the SAT instances generated by our algorithm in the experiments. The experimental results show the effectiveness of the random, estimating and greedy strategies. Compared to the state-of-the-art algorithm for generating SAT instances with predefined solutions, namely, M-hidden, our algorithm could be more effective in generating hard SAT instances.

1. Introduction

The satisfiability (SAT) problem is a widely known NP-complete problem. It plays an important role in computer science. Existing work on the SAT problem focuses mainly on solving the problem, and many SAT solvers have been proposed. Verifying the performance of SAT solvers requires a large number of SAT instances as a resource (SAT Competition, 2016). Random SAT instances have become one of the main parts of resources. Generation algorithms are required for producing random SAT instances that are properly hard to solve by current SAT solvers. Investigating the generation algorithms of random SAT instances requires us to explore the characteristics that induce the hardness of the SAT instances and the search strategies that are employed by the SAT solvers. Therefore, this process could inspire researchers to improve existing SAT solvers. Conversely, generation algorithms for hard SAT instances could also be used for privacy preservation (Esponda et al., 2004, 2007, 2009), information hiding (Esponda, 2008b), data security (Esponda et al., 2007, 2004; Esponda, 2008a), and authentication and recognition (Dasgupta & Azeem, 2008; Dasgupta & Saha, 2009; Bringer & Chabanne, 2010; Zhao et al., 2015b; Luo et al., 2019).

Thus far, several researchers have attempted to develop algorithms for generating random SAT instances. These algorithms can be divided into two classes according to whether their solutions are predefined or not. Several algorithms can randomly generate only those SAT instances that do not have predefined solutions, e.g., the 0-hidden algorithm (Achlioptas, 2001; Achlioptas et al., 2005; Jia et al., 2007). In addition, Ansótegui et al. applied complex network theories to SAT problems and showed that most industrial SAT instances have scale-free structures and high modularity. Based on this, they proposed several random instance models to generate industrial-like instances (Ansótegui et al., 2009; Giráldez-Cru & Levy, 2016; Giráldez-Cru & Levy, 2017). The SAT instances that are generated by these algorithms can be used to test the performance of complete SAT solvers, but they might be hard to use in testing the performance of incomplete SAT solvers (Achlioptas et al., 2005; Jia et al., 2007). When an incomplete SAT solver cannot solve the SAT instances that are generated by such algorithms, we cannot determine whether that incomplete solver has failed or the generated instances are unsatisfiable. Algorithms that have predefined solutions, e.g., 1-hidden algorithm (Achlioptas et al., 2005; Jia et al., 2007), 2-hidden algorithm (Achlioptas et al., 2005), and q-hidden algorithm (Jia et al., 2007), are more promising. The SAT instances that are generated by algorithms that have predefined solutions can be used

to test not only complete SAT solvers but also incomplete SAT solvers. Moreover, they can be easily used to protect data privacy and enhance data security (Esponda et al., 2007). In other words, by encoding confidential data or private data as predefined solutions, the above algorithms can generate hard SAT instances from the data. Next, the SAT instances will be stored and used instead of private data, and attackers should solve the instances if they want to obtain the private data; thus, the private data are protected because of the hardness of the SAT instances.

SAT problems exist in various real-world applications, some of which have more than one solution. The SAT instances that are derived from real-world applications usually have specific structures, but testing SAT solvers requires an entire family of benchmarks (Achlioptas et al., 2005; Jia et al., 2007). Therefore, it is important to propose algorithms for generating SAT instances that have multiple solutions. Currently, few algorithms for generating random SAT instances with multiple predefined solutions have been proposed. Designing generation algorithms for random SAT instances with multiple predefined solutions could be useful for exploring the relationship between the hardness of the random SAT instances and the number of solutions. Moreover, in some security applications, multiple records must be protected simultaneously. For example, a negative database (Esponda et al., 2009), which can be converted to an SAT instance (and vice versa), attempts to hide a whole database and support database operations. In these applications, generation algorithms for hard SAT instances that have multiple predefined solutions could be more suitable than those that have a single predefined solution. Moreover, by hiding multiple records simultaneously, these algorithms could also hide the number of records, and this property is quite important in some scenarios (Ateniese et al., 2011; Lindell et al., 2013; Bradley et al., 2016). Furthermore, these generation algorithms could be used to hide real private data together with some fake data to confuse attackers and enhance the security of the real private data. Liu, Luo and Yue (2015) proposed the M-hidden algorithm for generating random hard SAT instances that have multiple predefined solutions. The algorithm chooses a solution s among the predefined solutions as the central solution, which holds the minimum Hamming distance between s and the other solutions. Then, it generates SAT instances that satisfy their deduced hardness conditions according to s to mislead the local search strategy to search in the direction away from s. They proved that if the Hamming distance of any two of the predefined solutions is larger than half of the number of variables, they cannot generate the instance that satisfies the hardness conditions of two solutions simultaneously (even if it satisfies the hardness conditions of the other solutions). Therefore, their algorithm can generate only hard SAT instances that hide predefined solutions that are relatively close to each other (i.e., the Hamming distances between those solutions are limited).

In this paper, we propose the KHidden-M algorithm for generating random SAT instances that have multiple predefined solutions. Specifically, we propose three strategies for generating an initial SAT instance that is satisfied by all of the predefined solutions for the KHidden-M algorithm, and we also propose a strategy for adjusting the initial SAT instance if it cannot satisfy the hardness condition for the local search strategy. The first strategy is called the random strategy, which randomly generates clauses that are satisfied by all of the predefined solutions. The second strategy is called the estimating strategy, which attempts to estimate a set of probability parameters to control the distribution of different

types of clauses and allows the clauses to be adjusted more easily to satisfy the hardness condition (or directly satisfy the hardness condition) for the local search strategy. The third strategy is called the greedy strategy, which attempts to greedily generate clauses that can directly satisfy the hardness condition. Compared with the M-hidden algorithm (which is a state-of-the-art algorithm for generating hard SAT instances with multiple predefined solutions), our algorithm no longer considers the Hamming distance between the predefined solutions but makes the generated SAT instances satisfy the hardness condition of each solution as much as possible and iteratively optimizes it if it does not. Experimental results show that our algorithm could be more effective than the M-hidden algorithm. Specifically, the KHidden-M algorithm can generate harder SAT instances when the number of predefined solutions is large (e.g., not less than 10) or the Hamming distance between any two solutions is large (e.g., not less than half of the number of variables), and it can generate SAT instances that satisfy the hardness condition for the local search strategy from more general predefined solutions. Overall, our contributions are listed as follows.

(1) We propose an algorithm for generating random SAT instances that have multiple predefined solutions and are difficult to solve by the local search strategy.

(2) We propose three strategies to generate an initial SAT instance that is satisfied by all of the predefined solutions for the KHidden-M algorithm, i.e., the random, estimating and greedy strategies. The effectiveness of the three strategies is demonstrated in experiments.

(3) We investigate the hardness of solving the SAT instances generated by the proposed algorithm by two typical SAT solvers, i.e., WalkSAT (Selman et al., 1995) and Kissat (Fleury & Heisinger, 2020), and our experimental results show that our algorithm could be more effective than the M-hidden algorithm in terms of generating hard instances from more general predefined solutions.

The remainder of this paper is organized as follows. Section 2 presents related work on generation algorithms for random SAT instances. Section 3 introduces the concept of SAT instances and the hardness condition for the local search strategy. The basic KHidden-M algorithm and the random, estimating, and greedy strategies are given in Section 4. Section 5 shows the experimental results of using the SAT solvers to solve SAT instances generated by the KHidden-M algorithm. The KHidden-M algorithm is also compared with the M-hidden algorithm in Section 5. Some issues about the KHidden-M algorithm are discussed in Section 6. The proposed work is concluded in Section 7.

2. Related Work

Several algorithms for generating random SAT instances have been proposed. Specifically, the 0-hidden algorithm was first proposed to randomly generate SAT instances, and the instances (called 0-hidden instances for simplicity) are used to test SAT solvers (Achlioptas, 2001; Achlioptas et al., 2005; Jia et al., 2007). Selman et al. (1996) proposed using appropriate parameters and the right distribution to generate random SAT instances for which it is difficult to test the satisfiability; however, they did not consider the hiding of predefined multiple solutions. Achlioptas (2000, 2005) and Jia et al. (2007) showed that

the 0-hidden instances could not be used to test incomplete SAT solvers, and the 1-hidden algorithm was proposed to generate random SAT instances (called 1-hidden instances for simplicity) that have a predefined solution. Barthel et al. (2002) proposed a statistical mechanics approach based on the existence of a first order ferromagnetic phase transition and the glassy nature of excited states for generating hard and satisfiable 3-SAT instances. Achlioptas et al. (2005) showed that 1-hidden instances are easy to solve by some solvers such as WalkSAT (Selman et al., 1995), and they proposed the 2-hidden algorithm, which hides not only the predefined solution but also its complementary solution. They claimed that hiding the complementary solution can eliminate the bias of SAT instances toward the predefined solution, and their experimental results showed that the 2-hidden algorithm could generate SAT instances (called 2-hidden instances) that are harder to solve by WalkSAT than the 1-hidden instances. Jia et al. (2007) showed that the 2-hidden instances are also easy to solve by some solvers such as WalkSAT, and they proposed the q-hidden algorithm. The q-hidden algorithm generates three different types of clauses with different probabilities, and these probabilities are controlled by a parameter q. Their experimental results showed that the q-hidden algorithm can generate SAT instances (called q-hidden instances) that are hard to solve by the solvers based on the local search strategy (Selman et al., 1995), e.g., WalkSAT, because these solvers are guided in a wrong direction when searching for solutions. The q-hidden instance has one predefined solution.

Liu, Luo and Yue (2014) improved the q-hidden algorithm and proposed the p-hidden algorithm. The p-hidden algorithm is used to generate negative databases, which is a new privacy-preserving technique. Esponda et al. (2007) demonstrated that every negative database can be converted to an SAT instance, and vice versa. Thus, the p-hidden algorithm can also be used to generate SAT instances, and each instance has one predefined solution. The p-hidden algorithm uses parameters p_1 and p_2 to control the probabilities of generating different types of clauses in an SAT instance. The p-hidden algorithm can search over a wider scope than the q-hidden algorithm to generate harder SAT instances. Zhao et al. (2015a, 2017) proposed a more fine-grained algorithm called the K-hidden algorithm to generate hard-to-reverse negative databases. Compared with the p-hidden algorithm and q-hidden algorithm, their algorithm can generate harder K-SAT instances, and it can control the hardness of the SAT instances (against the local search strategy) in a more fine-grained manner. The K-hidden algorithm also has only one predefined solution. Xu et al. (2000, 2006) proposed several models for generating random constraint satisfaction instances, and they showed that their models can also be used to generate satisfiable and hard SAT instances (Xu et al., 2007). They claimed that their models can precisely locate a threshold where all of the instances are hard to solve, but they did not investigate the hiding of multiple predefined solutions. Oleksii Omelchenko et al. (2021) proposed a configurable model that first samples each variable based on the number of occurrences and then generates a random instance with the prescribed degrees. Giráldez-Cru et al. (2021) proposed the popularity-similarity model for random SAT instances, which can generate formulas with a power-law distribution for the number of variable occurrences and high clustering between them. They showed that the performance of CDCL SAT solvers is related to the existence of popularity and similarity in the SAT formulas. Barak-Pelleg (2022) et al. proposed a random industrial SAT model, which focuses on the community structure of instances, and

they showed experimentally that the satisfiability threshold of such instances tends to be lower than that for random SAT and even vanishes under certain conditions.

Liu, Luo and Yue (2015) proposed an algorithm called the M-hidden algorithm for generating random SAT instances with multiple predefined solutions. However, to successfully generate SAT instances that are expected to be hard to solve by the local search strategy, the predefined solutions in their algorithm should be somewhat close to each other (e.g., every predefined solution should have more than half of the assignments being the same as any of the other predefined solutions). This algorithm is adopted for experimental comparisons with the algorithm proposed in this paper.

In this paper, we propose the KHidden-M algorithm, which could be more effective than the M-hidden algorithm, to generate random hard SAT instances with multiple predefined solutions. Note that our algorithm can also be used to generate hard-to-reverse negative databases.

3. Preliminaries

In this section, we introduce the concept of the SAT instance (also called the SAT formula) in Subsection 3.1, and we introduce the hardness condition for the local search strategy in Subsection 3.2.

3.1 SAT Instance

In general, one SAT instance is a conjunctive normal form (CNF) formula with n logical variables $v_1, ..., v_n$, each variable has two literals, a positive literal l_i $(i = 1, ..., n)$ and a negative literal $\bar{l_i}$. The positive literal l_i is satisfied if and only if the variable v_i is assigned "True", and the negative literal $\bar{l_i}$ is satisfied if and only if the variable v_i is assigned "False". Specifically, SAT instance C consists of the conjunction of m clauses $c_1, ..., c_m$, each of which is the disjunction of several literals. C is satisfied if and only if all the clauses in C are satisfied, and a clause is satisfied if and only if at least one literal in the clause is satisfied. A satisfying assignment for C is a set of assignments to variables $v_1, ..., v_n$, and this satisfying assignment is called one solution of C. For example, if $C = (l_1 \vee \bar{l_2} \vee l_3) \wedge (l_1 \vee \bar{l_3} \vee l_5) \wedge (\bar{l_3} \vee \bar{l_4} \vee \bar{l_5})$, then $n = 5$, $m = 3$ and a solution is $v_1 =$ true, $v_2 =$ false, $v_3 =$ false, $v_4 =$ true and $v_5 =$ true. For simplicity, we denote "True" 1 and "False" 0 in the rest of this paper, and the above assignments can be denoted a string 10011, corresponding to variables $v_1, ..., v_5$, respectively. SAT problems exist in various real-world applications, and the problems are extracted as SAT instances while their solutions are unknown but important. Thus, SAT solvers are designed to solve the SAT instances to obtain the solutions.

3.2 Hardness Condition for the Local Search Strategy

The local search strategy is widely used in SAT solvers to search the solutions of SAT instances. The well-known solver WalkSAT (Selman et al., 1995) based on the local search strategy is described in Algorithm 1 below. First, the algorithm generates a set of assignments for all variables in step 2. Then, steps 3−18 attempt to flip variables (i.e., switches "True" to "False" or "False" to "True") to find a solution for C. At each iteration, an un-

satisfied clause c is randomly chosen first. Then if there is a variable in c with *break-count* (i.e., the number of currently satisfied clauses that become unsatisfied after flipping) equals 0, this variable is flipped. Otherwise, with probability p, the algorithm flips a random variable in c, and with probability $1-p$, it flips a variable in c that minimizes *break-count*. Such flips can be performed at most $maxFlips$ times. Once a solution for C is found, it is immediately returned to step 5. This process can be repeated up to $maxTries$ times, and then, if no solution is found, the algorithm returns a failure message in step 20.

Algorithm 1 WalkSAT

Input: An SAT instance C, $maxTries$, $maxFlips$, noise parameter $p \in [0,1]$
Output: A satisfying assignment for C, or FAIL

```
1: for i ← 1 to maxTries do
2:   σ ← a randomly generated truth assignment for C
3:   for j ← 1 to maxFlips do
4:     if C is satisfied by σ then
5:       return σ
6:     end if
7:     c ← a randomly selected unsatisfied clause in C
8:     if there is a variable x in c with break-count = 0 then
9:       v ← x
10:    else
11:      if random(0,1) ≤ p then
12:        v ← a randomly selected variable in c
13:      else
14:        v ← a variable in c with the smallest break-count
15:      end if
16:    end if
17:    Flip v
18:  end for
19: end for
20: return FAIL
```

The hardness condition for the local search strategy has been analyzed in some work (Achlioptas, 2001; Achlioptas et al., 2005; Jia et al., 2007; Liu et al., 2015, 2014; Zhao et al., 2015a). Specifically, assume that s is a solution for C, and the clauses in C are divided into K types according to s, where the type i ($i = 1, ..., K$) clause has i literals that are satisfied by s. Assume that p_i refers to the proportion of type i clauses to all clauses in C. As shown in Subsection 3.1, s can be denoted an n-bit string. For another random string t, if it has u bits that are different from s, the probability that a type i clause is unsatisfied by t can be calculated as follows (Jia et al., 2007; Liu et al., 2014; Zhao et al., 2015a):

$$P_{i,t} = \frac{\binom{n-K}{u-i}}{\binom{n}{u}} = \frac{\prod_{j=0}^{K-i-1}(n-u-j) \times \prod_{j=0}^{i-1}(u-j)}{\prod_{j=0}^{K-1}(n-j)} \xrightarrow{n\to\infty} \alpha^i \times (1-\alpha)^{K-i},$$

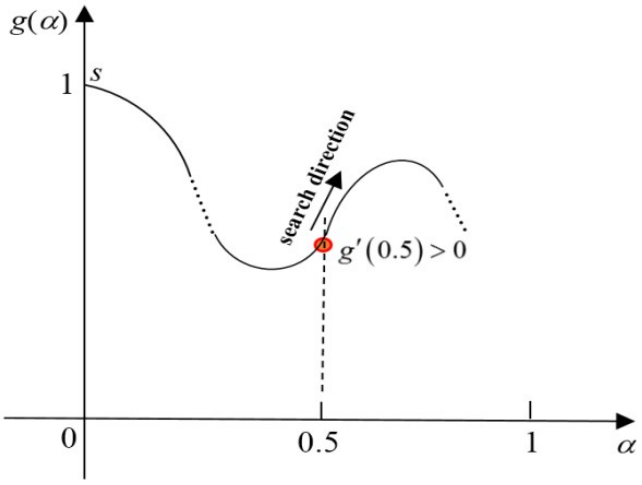

Figure 1: Example of searching s using the local search strategy.

where $\alpha = u/n$. Next, the expected proportion of clauses in C that are satisfied by t can be calculated as follows (Jia et al., 2007; Liu et al., 2014; Zhao et al., 2015a):

$$g(\alpha) = 1 - \sum_{i=1}^{K} p_i \times \alpha^i \times (1-\alpha)^{K-i}.$$

Jia et al. (2007) concluded through theoretical and experimental analysis that when $g'(0.5) > 0$, it is difficult for the local search strategy to find solution s. Figure 1 presents an example of searching s by the local search strategy. The horizontal coordinate is the value of α between the current string t (i.e., the current assignments that are achieved by the local search strategy) and solution s. The vertical coordinate is the expected proportion of clauses that are satisfied by t. Initially, t is expected to have approximately $0.5n$ bits different from s (when n is large enough, e.g., $n > 100$) because the strings that have approximately $0.5n$ bits different from s occupy more than half of the search space. According to the mechanism of the local search strategy, it will search along the direction of satisfying more clauses in C, i.e., the direction away from s. Therefore, when $g'(0.5) > 0$, the local search strategy is expected to be unable to find s. Based on $g(\alpha)$, $g'(0.5) > 0$ is converted to the following (Liu et al., 2014; Zhao et al., 2015a):

$$f = g'(0.5) = \sum_{j=1}^{K} (K - 2 \times j) \times p_j > 0. \tag{1}$$

Formula (1) is considered the hardness condition for the local search strategy.

4. KHidden-M Algorithm

In this section, the KHidden-M algorithm is proposed. Subsection 4.1 describes the algorithm framework and the random strategy, and Subsection 4.2 gives the details of the

adjustment strategy. The estimating strategy and the greedy strategy for improving the basic KHidden-M algorithm are presented in Subsections 4.3 and 4.4, respectively.

4.1 Basic Algorithm and Random Strategy

The KHidden-M algorithm is shown in Algorithm 2. The input includes a set of predefined solutions $S = \{s_1, ..., s_{n_s}\}$ and the parameters K and r. Parameter K determines that the generated clauses will have exactly K literals (these clauses are called K-clauses for simplicity), and parameter r is the ratio of the number of clauses to the number of variables, i.e., $m = n \times r$, where n is the number of variables. The output is the SAT instance C that consists of m generated clauses. Steps 1−6 constitute the random strategy.

Algorithm 2 KHidden-M algorithm

Input: A set of solutions $S = \{s_1, ..., s_{n_s}\}$, K, r
Output: An SAT instance C
1: $C \leftarrow \emptyset$, $m \leftarrow n \times r$
2: $i \leftarrow 0$
3: **while** $i < m$ **do**
4: $t \leftarrow$ a randomly generated clause that is satisfied by all of the solutions in S
5: $C \leftarrow C \wedge t$, $i \leftarrow i + 1$
6: **end while**
7: **if** $isHard(C, S) =$ False **then**
8: **return** $adjust(C, S)$
9: **end if**
10: **return** C

In the KHidden-M algorithm, first, C is initialized as the empty set, and m is set to be $n \times r$. Next, m clauses, which are satisfied by all of the solutions in S, are generated by m iterations and added to C in steps 2−6. Specifically, in step 4, clause t is generated as follows: 1) K different variables are randomly selected uniformly and independently; 2) all of the clauses that contain only the literals of the selected K variables are enumerated and are satisfied by all of the solutions in S; and 3) a clause is randomly selected from the clauses obtained in 2). Note that the time complexity of step 4 is $O(max(2^K, n_s \times K))$. Then, step 7 checks whether C is expected to be hard to solve by the local search strategy using the function $isHard(C, S)$. This function is based on the hardness condition for the local search strategy, and it returns "true" only when the following conditions are satisfied simultaneously:

$$\begin{cases} \sum_{j=1}^{K}(K - 2 \times j) \times p_{1,j} > 0 \\ \vdots \\ \sum_{j=1}^{K}(K - 2 \times j) \times p_{i,j} > 0 \\ \vdots \\ \sum_{j=1}^{K}(K - 2 \times j) \times p_{n_s,j} > 0 \end{cases}, \tag{2}$$

where $P = \{P_1, ..., P_{n_s}\}$, $P_i = \{p_{i,0}, ..., p_{i,K}\}(i = 1, ..., n_s)$, and $p_{i,j}(j = 0, ..., K)$ is the proportion of the type j clause with regard to s_i in C. With regard to s_i, the K-clauses can be divided into the following $K + 1$ types:

- ***Type 0 clause*** $T_{i,0}$: All of its literals are unsatisfied by s_i.
  ⋮
- ***Type*** j ***clause*** $T_{i,j}$: It has j literals that are satisfied by s_i, and the others are unsatisfied by s_i.
  ⋮
- ***Type*** K ***clause*** $T_{i,K}$: All of its literals are satisfied by s_i.

If C does not satisfy the hardness condition in (2), then it will be adjusted in step 8, and the adjusted C will be returned. Otherwise, C will be returned in step 10. The details of the function *adjust()* are given in Subsection 4.2, and *adjust()* returns the adjusted instance and a Boolean value to identify whether the adjustment is successful. Although the returned Boolean value is not used here, we collect it as experimental results in Section 5. Assume that n_s predefined solutions have t_i situations for the value for the i-th combination of K variables selected from n variables, so the probability of generating two duplicate clauses in the KHidden-M algorithm is $\frac{1}{\binom{n}{K}^2} \times \sum_{i=1}^{\binom{n}{K}} \frac{1}{(2^K - t_i)^2}$, which is very low; thus we generally do not consider it. If and only if the number of given predefined solutions is larger than $2^n - 2^{n-K}$, the KHidden-M algorithm cannot generate the instance that satisfies all solutions and will fall into an infinite loop (which can be avoided by simply adding corresponding conditional judgments to the code).

4.2 Adjustment Strategy

The details of the adjustment strategy are given in Algorithm 3. The input includes C, S, the maximum number of iterations $maxIter$ and ϵ used to control the data precision. Generally, we set $maxIter$ and ϵ to 3000 and 10^{-6}, respectively. The output is the adjusted instance and a Boolean value. The Boolean value is true if C is successfully adjusted to satisfy (2); otherwise, it is false.

In the 1st step, the f values of all solutions in S are calculated by the function *getF()*, where the f value of $s_i(i = 1, ..., n_s)$ is defined as follows, and $F = \{f_1, ..., f_{n_s}\}$. Note that the function *getF()* has no effect on the current SAT instance.

$$f_i = \sum_{j=1}^{K} (K - 2 \times j) \times p_{i,j}. \tag{3}$$

Algorithm 3 $adjust(C, S, maxIter, \epsilon)$

Input: An SAT instance $C = c_1 \wedge, ..., \wedge c_m$, a set of solutions $S = \{s_1, ..., s_{n_s}\}$, $maxIter$, ϵ
Output: Adjusted instance C, True/False

1: $F \leftarrow getF()$
2: **for** $j \leftarrow 1$ to $maxIter$ **do**
3: $idx \leftarrow$ a random permutation of $\{1, ..., m\}$
4: **for** $i \leftarrow 1$ to m **do**
5: $temp\_c \leftarrow c_{idx[i]}$
6: $flip\_n \leftarrow random(\{1, ..., K\})$
7: $B = \{b_1, ..., b_{flip\_n}\} \leftarrow$ randomly select $flip\_n$ literals from $c_{idx[i]}$
8: $c_{idx[i]} \leftarrow (c_{idx[i]} \setminus B) \cup \{\bar{b_k} | b_k \in B\}$
9: **if** $c_{idx[i]}$ is satisfied by all of the solutions **then**
10: $F' \leftarrow F$
11: $v \leftarrow dF(c_{idx[i]}, temp\_c)$
12: **if** $v < -\epsilon$ **then**
13: $c_{idx[i]} \leftarrow temp\_c$, $F \leftarrow F'$
14: **else if** $-\epsilon \leq v \leq \epsilon$ **then**
15: $c_{idx[i]} \leftarrow temp\_c$, $F \leftarrow F'$ with probability 50%
16: **else if** $isHard(C) =$ True **then**
17: **return** C **and True**
18: **end if**
19: **end if**
20: **end for**
21: **end for**
22: **return** C **and False**

Steps 2−21 attempt to randomly flip the literals in C to iteratively enhance the f values of the predefined solutions. In each iteration, a random permutation of $\{1, ..., m\}$ is generated first, and this permutation determines the order of the clauses in which literals will be flipped. For the i-th selected clause, after it is backed up to $temp\_c$ in step 5, $flip\_n$ literals will be randomly chosen and flipped in steps 6−8, where $flip\_n$ is a random number that is generated in step 6. Assume that $T_i(x)$ denotes the type of clause x with regard to s_i, and $dF(x, y)$ for any clause x or y denotes the gain after substituting x for y instead of simply the sum of changes in the f value of the predefined solutions. After flipping, if the new clause is still satisfied by all of the solutions in S, then $dF(c_{idx[i]}, temp\_c)$ will be computed to judge whether this flip is useful in steps 10−19. Specifically, if C does not satisfy the hardness condition in (2) before or after flipping, $dF(x, y)$ can be calculated as follows:

$$dF(x,y) = \sum_{i=1}^{n_s} \left(\left[(f_i < 0) \vee \left(f_i + \frac{2\times(T_i(y)-T_i(x))}{m} < 0 \right) \right] \times \frac{2\times(T_i(y)-T_i(x))}{m} \right) , \quad (4)$$

where $\left[(f_i < 0) \vee \left(f_i + \frac{2\times(T_i(y)-T_i(x))}{m} < 0 \right) \right]$ returns 1 when it is true; otherwise, it returns 0.

If C satisfies the hardness condition in (2) both before and after flipping, then $dF(x, y)$ can be calculated as follows:

$$dF(x, y) = min\{f_i^{'}|\forall i = 1, ..., n_s\} - min\{f_i|\forall i = 1, ..., n_s\}, \tag{5}$$

where $min\{f_i|\forall i = 1, ..., n_s\}$ and $min\{f_i^{'}|\forall i = 1, ..., n_s\}$ denote the minimum f values before and after flipping, respectively. If the minimum f value before flipping is equivalent to the minimum f value after flipping, then $dF(x, y)$ will be recalculated as follows:

$$dF(x, y) = \sum_{i=1}^{n_s} \frac{2 \times (T_i(y) - T_i(x))}{m}. \tag{6}$$

If $dF(x, y)$ is less than $-\epsilon$, the flipping and f values will be restored in step 13. If $|dF(x, y)|$ is not larger than ϵ, the flipping and f values will be restored with a probability of 50% in step 15. Otherwise, flipping will take effect, and if C satisfies (2), then Algorithm 3 will immediately return the adjusted C and "True" in step 17. If C does not satisfy (2) after $maxIter$ iterations, Algorithm 3 will return the adjusted C and "False" in step 22 (which means that even if the final adjustment fails, the previous useful flips will still be saved). Note that the value of $maxIter$ can be set empirically or according to pretesting. Usually, when $maxIter$ is larger, Algorithm 3 can be more effective in adjusting C to satisfy (2), but more computational cost are needed. According to the computing resources, users can properly enlarge the value of $maxIter$ to enhance the success rate of the adjustment strategy.

4.3 Estimating Strategy

In this subsection, we provide a strategy called the estimating strategy to improve the basic KHidden-M algorithm. With this improvement, the rate of successfully generating a random SAT instance that satisfies (2) could be enhanced (as demonstrated in Section 5).

Algorithm 4 Estimating strategy

Input: A set of solutions $S = \{s_1, ..., s_{n_s}\}$, K, r
Output: An SAT instance C

1: $C \leftarrow \emptyset$, $m \leftarrow n \times r$
2: $s_x \leftarrow$ randomly select a solution from S
3: Calculate $P_x = \{p_{x,0}, ..., p_{x,K}\}$ according to (12) or (13)
4: Let $SP = \{SP_0, ..., SP_K\}$, where $SP_0 = 0$ and $SP_i = p_{x,0} + ... + p_{x,i} (i = 0, ..., K)$
5: $i \leftarrow 0$
6: **while** $i < m$ **do**
7: $\quad temp\_v = random(0, 1)$
8: $\quad t \leftarrow$ randomly generate a type u clause (with regard to s_x) that is satisfied by all of the solutions in S, where u satisfies $SP_{u-1} \leq temp\_v < SP_u$
9: $\quad C \leftarrow C \wedge t$, $i \leftarrow i + 1$
10: **end while**
11: **return** C

Literals		**Type with regard to s_x**				
Part 1	**Part 2**	0	...	u	...	K
0	K	$T_{y,K}$	...	$T_{y,K-u}$	...	$T_{y,0}$
1	$K-1$	$T_{y,K-1}$	...	$\frac{u}{K}T_{y,K-u+1} + \frac{K-u}{K}T_{y,K-u-1}$	...	$T_{y,1}$
...	...	...	...	...	...	...
w	$K-w$	$T_{y,K-w}$	...	$\sum_{v=0}^{min\{w,u\}} \frac{\binom{w}{v}\times\binom{K-w}{u-v}}{\binom{K}{u}} T_{y,K-w-u+2v}$	...	$T_{y,w}$
...	...	...	...	...	...	...
K	0	$T_{y,0}$	...	$T_{y,u}$	...	$T_{y,K}$

Table 1: Proportions of different types of clauses with regard to s_y

In the random strategy in Algorithm 2, the initial m clauses of instance C are arbitrarily and randomly generated (while being satisfied by all of the predefined solutions). In this case, the f values of the predefined solutions that are obtained from C are usually less than 0. Therefore, the KHidden-M algorithm would need much effort or even fail when it attempts to adjust C to satisfy (2). Thus, we present the estimating strategy for generating the initial C in Algorithm 4. The estimating strategy exploits the fact that if the SAT instance is generated according to a predefined $P_i = \{p_{i,0}, ..., p_{i,K}\}(p_{i,0} = 0)$ for a solution s_i, the P_j of any other predefined solution s_j can be roughly estimated (Liu et al., 2015). Thus, by selecting an appropriate P_i, the estimating strategy roughly controls P_j and attempts to generate a random SAT instance that may be easier to adjust to satisfy the hardness condition (or directly satisfy the hardness condition). Specifically, in Algorithm 3, a solution s_x is randomly selected from S in step 2, and an appropriate P_x is computed in step 3. In steps 6−10, m iterations are performed. In each iteration, the type $j(j = 1, ..., K)$ clause with regard to s_x is generated with probability $p_{x,j}$, and then, the generated clause is added to C in step 9. The details of obtaining an appropriate P_x are described as follows.

First, we consider the case where there are two predefined solutions, i.e., s_x and s_y. We divide n variables into two parts, i.e., part 1 contains the variables where s_x and s_y have the same assignments, and part 2 contains the variables where s_x and s_y have different assignments. Assume that part 1 contains u variables, that $u = \alpha \times n$ and that part 2 contains $n - u$ variables. When generating an SAT instance C that has the solution s_x and is hard to solve by the local search strategy, the proportions of clauses in C are expected to satisfy $f_x = \sum_{j=1}^{K}(K - 2 \times j) \times p_{x,j} > 0$ and $p_{x,0} = 0$. In step 8, when randomly generating the clause t, K variables are selected first. The probability that t contains w variables that belong to part 1 and $K - w$ variables that belong to part 2 is as follows (when $n \to +\infty$) :

$$q_w = \binom{K}{w} \times \alpha^w \times (1 - \alpha)^{K-w}. \tag{7}$$

Table 1 shows the proportions of different types of clauses with regard to s_y when t is a type $u(u = 1, ..., K)$ clause with regard to s_x(without considering whether t is satisfied by s_y). $T_{y,j}$ in Table 1 denotes a type j clause with regard to s_y.

If solution s_y is also hidden, then the *type 0* clause with regard to s_y will be rejected in step 8 in Algorithm 4. According to (7) and Table 1, the expected proportion of the type j clause with regard to s_y can be calculated by (8). Note that the process of deducing $p_{y,j}$ is based on (Liu et al., 2015).

$$\begin{aligned} p_{y,j} &= \sum_{u=1}^{K} Psum_{y,u}^{-1} \times \sum_{w=0}^{K} Ps_{u,w}^{-1} \times \binom{K}{w} \times \alpha^w \times (1-\alpha)^{K-w} \times \\ &\quad \sum_{v=0}^{min\{w,u\}} \frac{\binom{w}{v} \times \binom{K-w}{u-v}}{\binom{K}{u}} \times p_{x,u} \times [j = K - w - u + 2v] \\ &= \sum_{u=1}^{K} Psum_{y,u}^{-1} \times \sum_{w=0}^{K} Ps_{u,w}^{-1} \times \alpha^w \times (1-\alpha)^{K-w} \times \\ &\quad [j - K + w + u \quad is \quad even] \times \binom{u}{\frac{j-K+w+u}{2}} \times \binom{K-u}{\frac{K+w-u-j}{2}} \times p_{x,u} \end{aligned}, \tag{8}$$

where $Ps_{u,w}$ $(u = 1, ..., K,\ w = 0, ..., K)$ is calculated as follows:

$$\begin{aligned} Ps_{u,w} &= 1 - \sum_{v=0}^{min\{w,u\}} \frac{\binom{w}{v} \times \binom{K-w}{u-v}}{\binom{K}{u}} \times [K - w - u + 2v = 0] \\ &= 1 - [K - w - u \quad is \quad even] \times \frac{\binom{w}{\frac{w+u-K}{2}} \times \binom{K-w}{\frac{K+u-w}{2}}}{\binom{K}{u}} \end{aligned}, \tag{9}$$

$Psum_{y,u}(u = 1, ..., K)$ is calculated as follows:

$$Psum_{y,u} = 1 - \sum_{w=0}^{K} \binom{K}{w} \times \alpha^w \times (1-\alpha)^{K-w} \times [Ps_{u,w} = 0]. \tag{10}$$

Dividing by $Ps_{u,w}$ in (8) occurs because when generating a type u clause with regard to s_x and the K variables (used for constructing the clause) are selected, the probability of generating other types of clauses with regard to s_y will be increased if the type 0 clause with regard to s_y is rejected. Dividing by $Psum_{y,u}$ occurs because if all of the type u clauses (with regard to s_x) that are constructed by the literals of the selected K variables are type 0 clauses with regard to s_y, these clauses will be rejected and the probability of selecting other sets of variables will be increased.

If C hides the n_s solutions in S, then the corresponding $\alpha_1, ..., \alpha_{n_s}$ between $s_1, ..., s_{n_s}$ and s_x can be calculated, where $\alpha_i(i = 1, ..., n_s)$ denotes the proportion of variables that have different assignments between s_i and s_x. Based on Algorithm 4 and $P_x = \{p_{x,1}, ..., p_{x,n_s}\}$, the expected proportion of type j clauses with regard to s_i can be roughly approximated as follows:

$$\begin{aligned} p_{i,j} \approx &\sum_{u=1}^{K} Psum_{i,u}^{-1} \times \sum_{w=0}^{K} Ps_{u,w}^{-1} \times \alpha^w \times (1-\alpha)^{K-w} \times \\ &[j - K + w + u \quad is \quad even] \times \binom{u}{\frac{j-K+w+u}{2}} \times \binom{K-u}{\frac{K+w-u-j}{2}} \times p_{x,u} \end{aligned}. \tag{11}$$

To generate an SAT instance C that hides $s_1, ..., s_{n_s}$ and is hard to solve by the local search strategy, (2) must be satisfied. Note that $p_{i,0} = 0 (i = 1, ..., n_s)$ because only the clauses that are satisfied by all of the solutions in S will be accepted and added to C (refer to Algorithm 4).

After substituting (11) into (2), we obtain n_s inequalities with K variables, $p_{x,1}, ..., p_{x,K}$. Several algorithms can be used to solve these inequalities and obtain an appropriate P_x, which can be used to generate SAT instances that are expected to be hard to solve by the local search strategy. For example, searching an appropriate P_x can be formalized as a linear programming problem in (12), and there are many algorithms for solving the linear programming problem. Note that the hardness of finding a solution s_i by the local search strategy is expected to increase with the f value of s_i (Jia et al., 2007; Liu et al., 2014; Zhao et al., 2017), and f_{linear} might be related to the overall hardness.

$$Minimize : f_{linear} = -\sum_{i=1}^{n_s}\sum_{j=1}^{K}(K - 2 \times j) \times p_{i,j}. \tag{12}$$

Subject to

$$\begin{cases} \sum_{j=1}^{K} p_{x,j} = 1 & \\ 0 \leq p_{x,j} \leq 1 & for \quad j = 1, ..., K \\ \sum_{j=1}^{K}(K - 2 \times j) \times p_{i,j} > 0 & for \quad i = 1, ..., n_s \end{cases},$$

where $p_{i,j}$ is calculated using (11).

When the above linear programming problem has a feasible solution, it can be effectively solved, and the expected $p_{x,1}, ..., p_{x,K}$ can be obtained. However, in a few cases, e.g., there is a solution in S that has more than 90% assignments different from s_x (i.e., $\alpha > 0.9$), the above linear programming problem might have no feasible solutions. In these cases, the 3rd constraint in (12) can be abandoned, and the modified linear programming problem can be solved to obtain a solution. Although this solution is not what we originally expected, it could be used to randomly generate an instance C that would be easier to adjust to satisfy the hardness condition in (2).

Another way, which could be more reasonable, to handle the case in which no feasible solutions exist is to redefine the object as follows:

$$Minimize : f_{quad} = \max_{i=1,...,n_s} \left\{ -\sum_{j=1}^{K}(K - 2 \times j) \times p_{i,j} \right\}. \tag{13}$$

The constraints are the first two constraints in (12), and the 3rd constraint is abandoned. Note that the above problem can be solved by the sequential quadratic programming (SQP) method (Boggs & Tolle, 1995), e.g., using the "*fminimax*" function in MATLAB.

After solving the problem, we can obtain a setting of $P_x = \{p_{x,0}, ..., p_{x,K}\}$, and we can randomly generate clauses of C according to P_x and s_x (as shown in Algorithm 4). According to the above analyses, the proportions of different types of clauses with regard to other solutions in S are expected to satisfy the hardness condition as well. Although the

generated SAT instance is expected to be hard to solve by the local search strategy, in a bad situation, it would violate some hardness conditions in (2) after the estimating strategy is called. In this situation, an adjustment strategy in Algorithm 3 will be performed to tune P. Even in the bad situation, the C generated based on the estimating strategy would be easier to adjust to satisfy the hardness condition in (2) than that generated based on the random strategy in Algorithm 2. The effectiveness of the estimating strategy is shown in Section 5.

4.4 Greedy Strategy

In this subsection, we provide a greedy strategy for improving the basic KHidden-M algorithm. Similar to the estimating strategy, this strategy attempts to improve the random generation of the initial C in steps 1−6 (i.e., the random strategy) in Algorithm 2. The greedy strategy is shown in Algorithm 5. By using this algorithm, we attempt to randomly generate an SAT instance that makes the f values of most of the predefined solutions larger than 0 and enhances the success rate of adjusting C to satisfy (2).

Algorithm 5 Greedy strategy

Input: A set of solutions $S = \{s_1, ..., s_{n_s}\}$, K and r
Output: An SAT instance C

1: $C \leftarrow \emptyset$, $m \leftarrow n \times r$
2: Initialize $F = \{f_1, ..., f_{n_s}\}$, $f_i \leftarrow 0 (i = 1, ..., n_s)$
3: $i \leftarrow 0$
4: **while** $i < m$ **do**
5: Let $S1$ be the set of predefined solutions that have the minimum f value in F
6: Randomly select a solution s_j from $S1$
7: $v \leftarrow random\left(1, ..., \lfloor \frac{K-1}{2} \rfloor\right)$
8: Randomly generate a type v clause t (with regard to s_j) that is satisfied by all of the solutions in S
9: $C \leftarrow C \wedge t, i \leftarrow i + 1$
10: $updateF(F, C, t, S)$
11: **end while**
12: **return** C

The greedy strategy attempts to make the current SAT instance satisfy (2) as much as possible when generating each clause. This may result in the generated SAT instance directly satisfying (2), rendering the adjustment strategy ineffective. In addition, the purpose of the greedy strategy is to make the generated SAT instance have the maximum f value with regard to each predefined solution, rather than just above zero. Intuitively, the f values of the SAT instance generated by the estimating strategy may be slightly greater than 0, while the f values of the SAT instance generated by the greedy strategy may be far greater than 0. The effectiveness of the greedy strategy is shown in Section 5.

In Algorithm 5, the f values in F are initialized to 0 in step 2. In steps 4−11, m iterations are performed to generate m clauses. In each iteration, a solution s_j that has the minimum f value in the current F is selected, and then, a type v clause t (with regard

to s_j) that is satisfied by all of the solutions in S is generated. Specifically, in step 8, t is generated as follows: 1) randomly select K variables; 2) enumerate all of the clauses that contain only the literals of the selected K variables and insert them into clause set L; 3) delete the clauses that are not satisfied by any solution in S or are not a type v clause with regard to s_j from L; and 4) randomly select a clause from L. Because v is randomly selected from the interval $\left[1, \lfloor \frac{K-1}{2} \rfloor\right]$, the f value of s_j will increase after t is added to C. The f values of all of the predefined solutions are updated by the function *updateF()* in step 10.

Algorithm 6 $updateF(F, C, t, S)$

Input: The set of f values $F = \{f_1, ..., f_{n_s}\}$, an SAT instance C, a clause t, and a set of solutions $S = \{s_1, ..., s_{n_s}\}$

1: $u \leftarrow$ the current number of clauses in C
2: **for** $i \leftarrow 1$ to n_s **do**
3: $\quad v \leftarrow$ the type of t with regard to s_i
4: $\quad f_i \leftarrow (u-1) \times \frac{f_i}{u} + \frac{K-2\times v}{u}$
5: **end for**

The details of $updateF()$ are given in Algorithm 6. Specifically, if t is a type v clause with regard to $s_i (i = 1, ..., n_s)$, then adding t to C leads to a change in $p_{i,v}$. Assume that $p_{i,j} (j = 1, ..., K)$ denotes the proportion of type j clauses with regard to s_i in C before adding t and that $p'_{i,j}$ denotes the proportion after adding t. Then, we have the following:

$$\begin{cases} p'_{i,j} = \frac{(u-1)\times p_{i,j}}{u}, & 1 \le j \le K \quad and \quad j \neq v \\ p'_{i,j} = \frac{(u-1)\times p_{i,j}+1}{u}, & j = v \end{cases}, \tag{14}$$

where u is the current number of clauses in C.

Assume that the f value of s_i before adding t is f_i and that the f value after adding t is f'_i. Thus, we have the following:

$$\begin{aligned} f'_i &= \sum_{j=1}^{K} (K - 2 \times j) \times p'_{i,j} \\ &= \sum_{j=1}^{K} (K - 2 \times j) \times \frac{(u-1)\times p_{i,j}}{u} + \frac{K-2\times v}{u} \\ &= \frac{u-1}{u} f_i + \frac{K-2\times v}{u} \end{aligned} . \tag{15}$$

The greedy strategy can be used to replace the random strategy (i.e., steps 1−6) in Algorithm 2. Section 5 shows that the greedy strategy can enhance the success rate of generating an SAT instance that satisfies (2) from multiple predefined solutions.

5. Experimental Results

In this section, we conduct several experiments to show the performance of the KHidden-M algorithm and the hardness of the instances that are generated by the KHidden-M algorithm (called KHidden-M instances for simplicity).

5.1 Experimental Setup

In the experiments, first, we randomly generate an n-bit string s_0, and next, we construct the predefined solutions in S based on s_0. Specifically, the assignment "True" is denoted 1, and "False" is denoted 0. Each solution s_i $(i = 1, ..., n_s)$ in S is denoted an n-bit string. A parameter $maxH$ is used to control the Hamming distances between the predefined solutions. When constructing a solution in S, t bits are randomly selected first, and then, s_i is set to be the same as s_0 except for those t bits being different. For $n_s - 2$ predefined solutions, t is a random number that is not larger than $maxH/2$. For the other two predefined solutions s_u and s_v, t is fixed as $maxH/2$. Moreover, the t bits of s_u are forced to be different from those of s_v, i.e., there are $maxH$ different bits between s_u and s_v. In the above way, the Hamming distance between any two predefined solutions in S is limited by $maxH$, and $maxH$ is set to $n/2$ by default. After S is generated, we randomly choose s_x from S and collect the values of $\alpha_1, ..., \alpha_{n_s}$ between $s_1, ..., s_{n_s}$ and s_x. Then, we use the function "$fminimax$" (which uses the sequential quadratic programming method) in MATLAB to calculate an appropriate P_x for the estimating strategy.

Next, the KHidden-M algorithm is used to generate an SAT instance C from S. Then, the SAT solvers WalkSAT and Kissat are used to solve C, where the number of flips made by WalkSAT and the runtime of Kissat are recorded. WalkSAT is set to restart at most 10^4 times and to perform at most 10^4 flips per time, and the *noise* is set to 0.5. Kissat is set to "sat" mode, and the maximum runtime is set to 3600 seconds. When the solver performs more flips or executes for a longer time, C is harder to solve by the solver. The above experiment is conducted 29 times, and the number of flips made by WalkSAT and the runtime of Kissat are recorded to evaluate the hardness of KHidden-M instances. Note that WalkSAT is a typical SAT solver based on the local search strategy and is widely used to verify the hardness of SAT instances (Achlioptas et al., 2005; Jia et al., 2007; Liu et al., 2015, 2014; Zhao et al., 2017). Specifically, we choose the v56 version of WalkSAT released in May 2018. Kissat is a state-of-the-art CDCL solver that won the first place of the main track in SAT Competition 2020. For simplicity, the KHidden-M algorithms using the random, estimating, and greedy strategies are called the KHidden-M-rand, KHidden-M-estimate, and KHidden-M-greedy algorithms, respectively.

5.2 Hardness of KHidden-M Instances against WalkSAT and Kissat

In this subsection, we investigate the influence of the parameters $r, n, maxH$ and n_s on the hardness of the KHidden-M instance against WalkSAT and Kissat. Moreover, we compare the effectivenesses of the random, estimating and greedy strategies.

5.2.1 Varying r

To investigate the influence of the parameter r on the hardness of the KHidden-M instance against WalkSAT and Kissat, we conduct this experiment. Specifically, the parameter settings are $n = 500$, $maxH = n/2$, $n_s = 2$ and $K = 3$. We vary r from 1 to 10 with a step size of 0.5. In each attempt, three SAT instances are generated by the KHidden-M-rand, KHidden-M-estimate and KHidden-M-greedy algorithms. The three instances are then solved by WalkSAT and Kissat, and this process is repeated 29 times for both solvers.

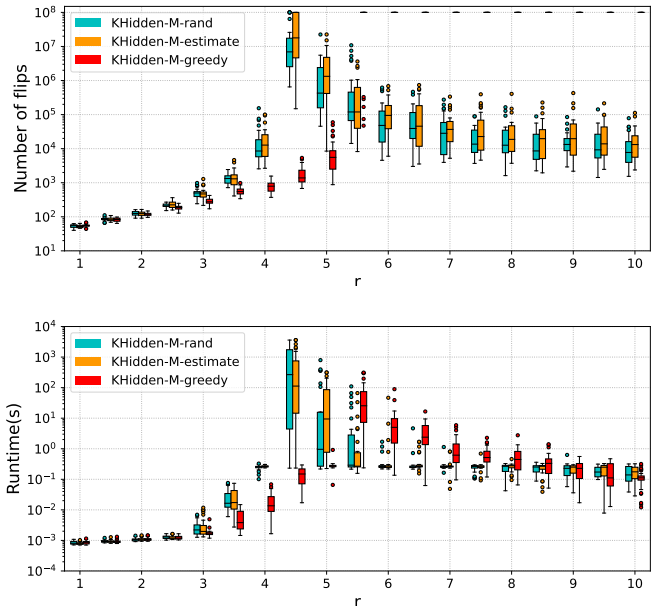

Figure 2: Performances of WalkSAT (top) and Kissat (bottom) in solving the SAT instances generated by the KHidden-M algorithm with the random, estimating and greedy strategies when r is varied from 1 to 10.

Note that in this experiment, regardless of whether a generated SAT instance satisfies the hardness condition in (2), it will be solved by two solvers.

Figure 2 (top) shows that when r is not larger than 4.5, the hardness for WalkSAT to solve the SAT instances generated by the KHidden-M-rand and KHidden-M-estimate algorithms increases rapidly with the increase in r. When r equals 4.5, WalkSAT needs approximately 20 million flips to solve the SAT instances generated by the KHidden-M-estimate algorithm but only approximately 10 million flips to solve the SAT instances generated by the KHidden-M-rand algorithm. The SAT instances generated by the KHidden-M-greedy algorithm are relatively easier to solve by WalkSAT than that generated by the other two algorithms when r is not larger than 5. The reason might be that all three algorithms can adjust the instances to satisfy the hardness condition; the KHidden-M-greedy algorithm generates instances with less randomness, and WalkSAT can solve the instances more easily. When r is not less than 4.5, the hardness of the SAT instances generated by the KHidden-M-rand and KHidden-M-estimate algorithms gradually decreases with increasing r. WalkSAT cannot successfully solve the SAT instances generated by the KHidden-M-greedy algorithm when r is not less than 5.5. All these results show that the estimating and greedy strategies can be more effective than the random strategy, and the greedy strategy can outperform the other two strategies when r is large (e.g., $r > 5.5$).

As shown in Figure 2 (bottom), regardless of how r changes, the performance of the KHidden-M-estimate algorithm is almost the same as that of the KHidden-M-rand algorithm. When r is less than 4.5, the hardness of the SAT instances generated by the KHidden-

M-rand and KHidden-M-estimate algorithms increases with increasing r and finally reaches a peak at r equals 4.5, which means that Kissat needs approximately 150 seconds to solve them. However, Kissat can solve the SAT instances generated by the KHidden-M-estimate and KHidden-M-rand algorithms in 1 second when r is less than 4. Similarly, the process of peaking the hardness of the SAT instances generated by the KHidden-M-greedy algorithm is delayed until r increases to 5.5, which means that Kissat takes approximately 40 seconds to solve them. When r is larger than 4.5, the hardness of the SAT instances generated by the KHidden-M-estimate and KHidden-M-rand algorithms decreases rapidly with increasing r and finally stabilizes at approximately 0.2 seconds. When r is larger than 5.5, the performance of the KHidden-M-greedy algorithm is the best, but it decreases rapidly with increasing r, and finally stabilizes at approximately 0.1 seconds. Overall, no matter how r is modified, the SAT instances generated by the three strategies are not difficult to solve by Kissat (most cases can be solved in seconds). These results show that for Kissat, the instances generated by our three algorithms have a certain hardness only when r is fixed to a specific value. Once r is modified, the hardness of the whole instance will drop rapidly.

5.2.2 Varying n

In this experiment, we investigate the influence of n on the hardness of the KHidden-M instance against WalkSAT and Kissat. In view of the results of r changes in the previous subsection, we decided to adopt the optimal r for different strategies: 4.5 for the random strategy and the estimating strategy and 5.5 for the greedy strategy. The settings of other parameters are $n_s = 2$, $maxH = n/2$ and $K = 3$. We vary n from 100 to 1500 with a step size of 100. The results are shown in Figure 3.

Figure 3 (top) shows that when n is not larger than 700, the hardness of the SAT instances (against WalkSAT) generated by the KHidden-M-rand and the KHidden-M-estimate algorithms increases with increasing n. When n is larger than 700, the number of flips made by WalkSAT to solve the KHidden-M-rand instances and the KHidden-M-estimate instances goes to the upper bound, i.e., 10^8. Increasing n usually leads to an exponential increase in the search space of the SAT solvers, and thus, the SAT solvers usually need to make more flips. The number of flips made by WalkSAT to solve the KHidden-M-greedy instances goes to the upper bound at a smaller n (i.e., 300), but the performance of the greedy strategy is more unstable than that of the other two strategies, a phenomenon gradually disappears when n is large enough (e.g., 1000). These results also show that with these parameter settings, the greedy strategy can be more effective (in generating hard SAT instances against WalkSAT) than the random and estimating strategies.

Figure 3 (bottom) shows that the performances of the three strategies are similar. Overall, the KHidden-M-estimate algorithm performs slightly better than the KHidden-M-rand and KHidden-M-greedy algorithms most of the time. When n is less than 800, the time spent by Kissat to solve the SAT instances generated by the three algorithms increases rapidly with increasing n, and finally reaches the upper limit, i.e., 3600 seconds. When n is not less than 800, Kissat can only solve several SAT instances generated by the three strategies within a specified time, and most of them are generated by the KHidden-M-greedy algorithm. These results show that the hardness of the SAT instances (against Kissat) gen-

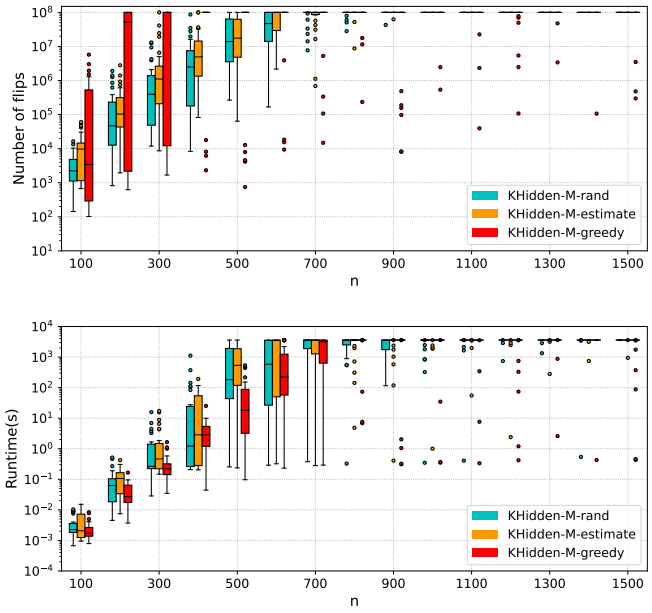

Figure 3: Performances of WalkSAT(top) and Kissat(bottom) in solving the SAT instances generated by the KHidden-M algorithm with the random, estimating and greedy strategies when n is varied from 100 to 1500.

erated by our three strategies can be guaranteed when the number of variables is sufficient and with other optimal parameters.

5.2.3 Varying $maxH$

In this experiment, we investigate the hardness of the KHidden-M instances generated based on the random, estimating and greedy strategies against WalkSAT and Kissat when varying $maxH$ from 0 to n. The results are shown in Figure 4. For simplicity, in Figure 4, we denote the KHidden-M algorithm that uses the adjust strategy with the random, estimating, and greedy strategies "rand-adjust", "estimate-adjust", and "greedy-adjust", respectively. We denote the KHidden-M algorithm that uses only the random, estimating, and greedy strategies (and does not use the adjustment strategy) "rand", "estimate", and "greedy", respectively. The parameter settings are $n = 500$, $r = 4.5$ for "rand", "rand-adjust", "estimate" and "estimate-adjust", $r = 5.5$ for "greedy" and "greedy-adjust", $n_s = 2$ and $K = 3$. For all of these settings, we use the same 29 predefined solution sets to generate SAT instances and record the median number of flips made by WalkSAT and the median Kissat runtime as results.

First, the performance of the adjustment strategy is examined. As Figure 4 shows, for the random and estimating strategies, the effect of using the adjustment strategy is obvious, while the improvement in the greedy strategy is relatively insignificant. The changes in the hardness of the SAT instances generated by the three strategies (against WalkSAT and

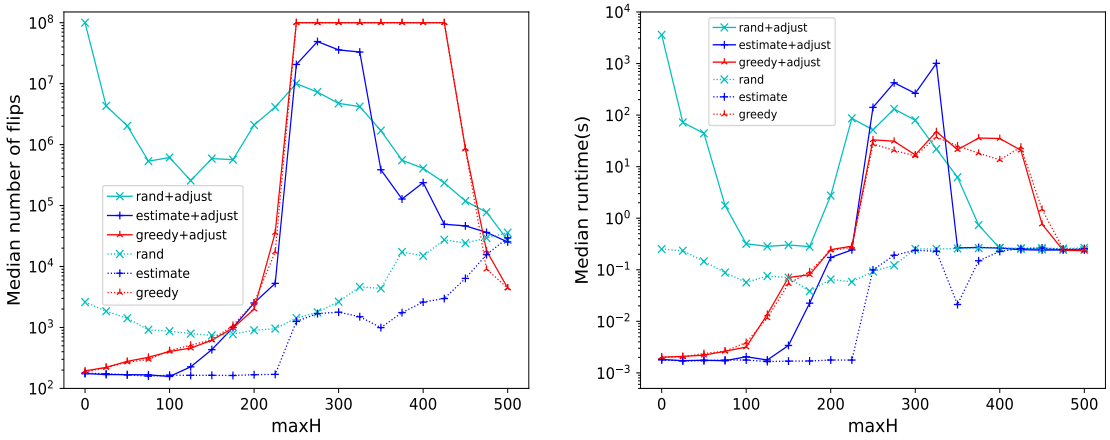

Figure 4: Performances of WalkSAT(left) and Kissat(right) in solving the SAT instances generated by the KHidden-M algorithm with random, estimating and greedy strategies when $maxH$ is varied from 0 to m.

Kissat) exhibit similar characteristics when $maxH$ varies. For the random strategy, the performance of the adjustment strategy is optimal when $maxH$ equals zero, but the effect rapidly decreases with the increase in $maxH$ when $maxH$ is less than 150. As $maxH$ continues to increase, the effect first increases and then decreases until no enhancement is made. For the estimating strategy, when $maxH$ is less than 350, the ability of the adjustment strategy to increase the hardness of the generated SAT instances improves as $maxH$ increases and then decreases rapidly as $maxH$ increases. For the greedy strategy, the plot is almost the same regardless of whether the adjustment strategy is used. We demonstrate through more experiments that these results are obtained because the greedy strategy can generate a random instance that initially satisfies the hardness condition in (2) and the adjustment strategy would not be called at all, while the random and estimating strategies exhibit the oppsite behavior, so the adjustment strategy can be used to further increase the hardness of the generated instances.

As shown in Figure 4 (left), when none of the three strategies use the adjustment strategy, the hardness of the SAT instances generated by the random strategy is always higher than that generated by the estimating strategy. The reason might be that the probability distribution added by the estimating strategy limits the randomness of the instance when $maxH$ is small (e.g., $maxH < n/2$). As $maxH$ increases to 250, WalkSAT is unable to solve the SAT instance generated by the greedy strategy under the limit of 10^8 flips. When $maxH$ is less than 225, the hardness of the SAT instance generated by the random strategy after using the adjustment strategy is much higher than that of the other two strategies. These results show that for WalkSAT, when $maxH$ is less than $n/2$, using the combination of the random strategy and the adjustment strategy is better. In other cases, using the greedy strategy with the adjustment strategy appears to be more effective.

When none of the three strategies uses the adjustment strategy, regardless of how $maxH$ changes, Kissat can solve the SAT instances generated by the random and estimating strate-

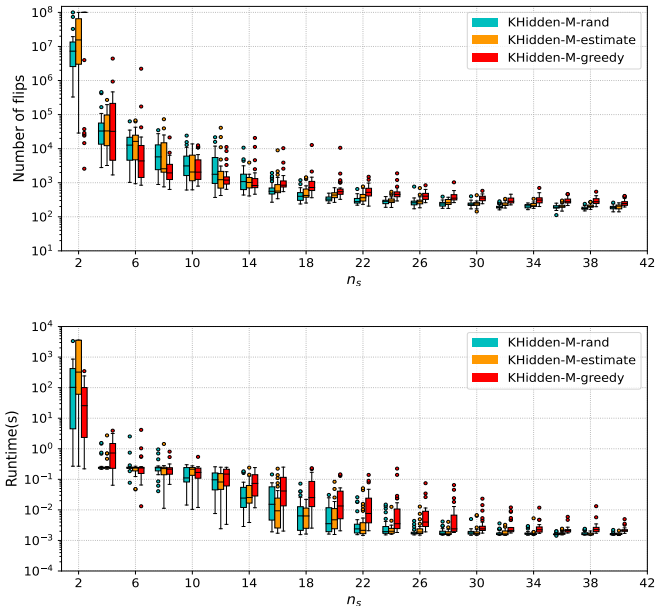

Figure 5: Performances of WalkSAT (top) and Kissat (bottom) in solving the SAT instances generated by the KHidden-M algorithm with random, estimating and greedy strategies when n_s is varied from 2 to 40.

gies in 1 second. When $maxH$ is between 250 and 425, Kissat needs approximately 30 seconds to solve the SAT instance generated by the greedy strategy. Figure 4 (right) shows that the most difficult instance (against Kissat) is generated by the random strategy after using the adjustment strategy when $maxH$ equals zero. After using the adjustment strategy, the estimating strategy performs best when $maxH$ is between 250 and 325, and the greedy strategy performs best when $maxH$ is larger than 325. Overall, when $maxH$ equals half of n, the performance of the three strategies and the adjustment strategy is the best.

Figure 4 also shows that when $maxH$ equals n (i.e., 500), the two predefined solutions are complementary to each other, and no SAT instances can satisfy the hardness condition in (2). Therefore, the adjustment strategy cannot modify the clauses to satisfy the hardness condition in (2), and it becomes useless in improving the hardness of the generated instances.

5.2.4 Varying n_s

In this experiment, we investigate the influence of the number of predefined solutions (i.e., n_s) on the hardness of the KHidden-M instances against WalkSAT and Kissat. We vary n_s from 2 to 40 with a step size of 2. The parameter settings are $n = 500$, $r = 4.5$ for the random and estimating strategies, $r = 5.5$ for the greedy strategy, $maxH = n/2$ and $K = 3$. The results are shown in Figure 5.

In Figure 5 (top), the plots of the three versions of the KHidden-M algorithm, which use the random, estimating and greedy strategies, are quite similar to each other. When n_s

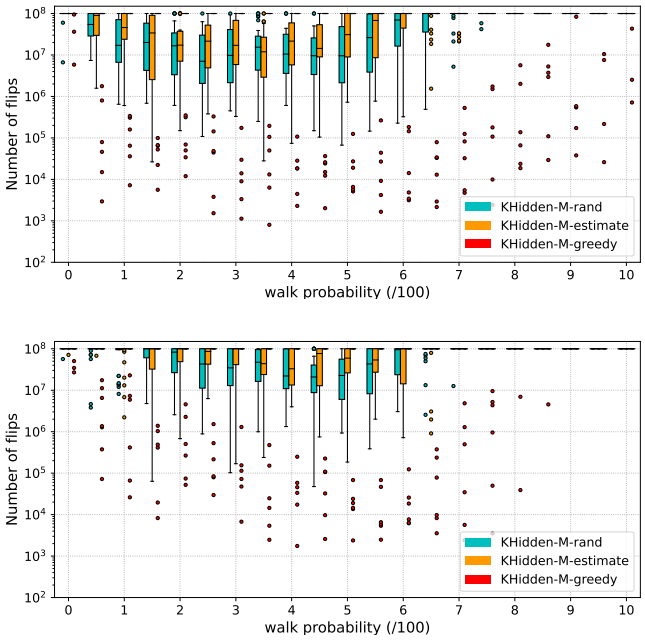

Figure 6: Performance of WalkSAT using the *best* (top) and *gsat* (bottom) heuristic strategies in solving the SAT instances generated by the KHidden-M algorithm with random, estimating and greedy strategies when *noise* is varied from 0 to 1.

equals 2, the KHidden-M-greedy algorithm performs better than the other two strategies, and WalkSAT cannot solve the generated SAT instances successfully under the limit of 10^8 flips. For the three algorithms, the number of flips made by WalkSAT roughly decreases with increasing n_s. This finding indicates that it would be more difficult to generate hard SAT instances against WalkSAT when there are more predefined solutions. It could be easier for SAT solvers to find one of the predefined solutions as the true assignment when there are more predefined solutions. In such case, it would be reasonable to adjust other parameters to generate hard SAT instances against WalkSAT, e.g., by enlarging n.

Similarly, Kissat shows the same trend as WalkSAT when solving the SAT instances generated by the KHidden-M algorithm. The time taken by Kissat to solve the SAT instances generated by the three strategies gradually decreases with increasing n_s. When n_s equals 2, the hardness of the SAT instance generated by the estimating strategy is higher than that generated by the other two strategies, and Kissat needs approximately 400 seconds to solve the instance. When n_s is not less than 4, the SAT instances generated by the three strategies can be solved by Kissat within 1 second, which means that we may need to modify other parameters to ensure the hardness of the instances.

5.2.5 Varying heuristics and the noise of WalkSAT

In this experiment, we investigate the influence of WalkSAT using different heuristic strategies and *noise* on solving KHidden-M instances. For the KHidden-M algorithm, the pa-

rameter settings are $n = 500$, $r = 4.5$ for the random and estimating strategies, $r = 5.5$ for the greedy strategy, $n_s = 2$, $maxH = n/2$ and $K = 3$. For WalkSAT, we choose two different heuristics, *best* (the default heuristic of WalkSAT) and *gsat*, which differ in the calculation of the greedy move. The *best* heuristic attempts to minimize the *break-count* (the number of unsatisfied clauses decreases after flipping) of each variable, while the *gsat* heuristic attempts to minimize the *break-count* of each variable minus *make-count* (the number of satisfied clauses increases after flipping) . For both heuristics, we vary the *noise* (which determines the probability of the random walk) from 0 to 1 with a step size of 0.05. The results are shown in Figure 6.

In Figure 6, WalkSAT using the *best* or *gsat* heuristic exhibits similar characteristics as the walk probability is changed. When solving the instance generated by the Khidden-M-rand and Khidden-M-estimate algorithms, the performance of both heuristics decreases first and then increases with the increase in walk probability. When the walk probability is between 0.2 and 0.5, the *best* heuristic shows optimal performance, and WalkSAT requires approximately 10 million flips to solve the SAT instances generated by the KHidden-M-rand and KHidden-M-estimate algorithms. The performance of the *gsat* heuristic is slightly worse; it is optimal when the walk probability is between 0.4 and 0.6, and WalkSAT needs approximately 40 million flips to solve the generated SAT instances. These results suggest that WalkSAT uses the *best* heuristic better than the *gsat* heuristic in solving the SAT instances generated by the KHidden-M algorithm.

Regardless of whether WalkSAT uses the *best* or *gsat* heuristic or how to adjust the *noise*, it cannot successfully solve the SAT instances generated by the KHidden-M-greedy algorithm under the limit of 10^8 flips. However, the greedy strategy is also the most unstable, and this is consistent with the results of the previous experiments on r and n. The result indicates that the SAT instances generated by the KHidden-M-greedy algorithm around the threshold (i.e., 5.5) maintain quite high hardness, and WalkSAT cannot move randomly to help itself approach any hidden predefined solution. Meanwhile, when the walk probability is greater than 0.6, neither of the two heuristics can successfully solve the SAT instances generated by our three algorithms. The reason for this result is that the high walk probability leads to the failure of heuristics (regardless of whether it is *best* or *gsat*), and WalkSAT wastes much time on meaningless random flipping variables. Overall, a good choice for WalkSAT is to use the *best* heuristic and to set the *noise* to 0.5 to solve the instance generated by the KHidden-M algorithm, which is also the default setting for WalkSAT.

5.3 Comparison with the M-hidden Algorithm

In this experiment, we compare the KHidden-M algorithm with the M-hidden algorithm in generating hard SAT instances. The default parameter settings are $n = 500$ and $r = 5.6$ (as in Liu et al., 2015). n_s is set to 2, 5, 10, 15 and 20, and $maxH$ is set to $n/4$, $n/2$, and $3n/4$. For the KHidden-M algorithm, K is set to 3, and the random, estimating, and greedy strategies are used. For the M-hidden algorithm, four settings of p_1 and p_2 are chosen, i.e., $(p_1 = 0.63, p_2 = 0.32)$, $(p_1 = 0.69, p_2 = 0.23)$, $(p_1 = 0.75, p_2 = 0.14)$ and $(p_1 = 0.81, p_2 = 0.05)$. These four settings of p_1 and p_2 are recommended in (Liu et al., 2015).

n_s	**2**	**5**	**10**	**15**	**20**
M-hidden ($p_1 = 0.63, p_2 = 0.32$)	4%	0	0	0	0
M-hidden ($p_1 = 0.69, p_2 = 0.23$)	26%	4%	7%	8%	5%
M-hidden ($p_1 = 0.75, p_2 = 0.14$)	84%	16%	40%	42%	34%
M-hidden ($p_1 = 0.81, p_2 = 0.05$)	**100**%	32%	67%	69%	72%
KHidden-M-rand	**100**%	**100**%	**100**%	**100**%	99%
KHidden-M-estimate	**100**%	**100**%	**100**%	**100**%	**100**%
KHidden-M-greedy	**100**%	**100**%	**100**%	**100**%	**100**%

Table 2: Success rates of the KHidden-M and M-hidden algorithms in generating instances that satisfy (2) when $maxH = n/4$.

n_s	**2**	**5**	**10**	**15**	**20**
M-hidden ($p_1 = 0.63, p_2 = 0.32$)	0	0	0	0	0
M-hidden ($p_1 = 0.69, p_2 = 0.23$)	0	0	0	0	0
M-hidden ($p_1 = 0.75, p_2 = 0.14$)	0	0	0	0	0
M-hidden ($p_1 = 0.81, p_2 = 0.05$)	0	0	0	0	0
KHidden-M-rand	**100**%	**100**	95%	26%	0
KHidden-M-estimate	**100**%	**100**%	**100**%	63%	3%
KHidden-M-greedy	**100**%	**100**%	**100**%	**91**%	**36**%

Table 3: Success rates of the KHidden-M and M-hidden algorithms in generating instances that satisfy (2) when $maxH = n/2$.

5.3.1 Success rate of generation

In this experiment, first, for each parameter setting, we randomly generate 100 predefined solution sets. Next, we use the M-hidden algorithm and the KHidden-M algorithm to generate SAT instances from the 100 predefined solution sets, and we compute their success rates in generating SAT instances that satisfy the hardness condition in (2).

The results for $maxH = n/4$ are shown in Table 2. The success rate of the M-hidden algorithm with $p_1 = 0.63$ and $p_2 = 0.32$ or $p_1 = 0.69$ and $p_2 = 0.23$ is lower than 30% for all of the settings of n_s. The success rate of the M-hidden algorithm is 100% only when $p_1 = 0.81, p_2 = 0.05$ and $n_s = 2$. For the other parameter settings in Table 2, the KHidden-M algorithm has higher success rates than the M-hidden algorithm. The success rate of the KHidden-M-rand algorithm is 100% for $n_s = 2, 5, 10$ and 15. For $n_s = 20$, its success rate is 99%. For all of the settings of n_s, the success rates of the KHidden-M-estimate and KHidden-M-greedy algorithms are 100%. These results show that the KHidden-M algorithm can be more effective than the M-hidden algorithm in generating SAT instances that satisfy the hardness condition in (2) when $maxH = n/4$.

The results for $maxH = n/2$ and $3n/4$ are shown in Tables 3 and 4, respectively. For all of the parameter settings in Tables 3 and 4, the M-hidden algorithm cannot generate an SAT instance that satisfies the hardness condition in (2). When $maxH = n/2$ and

n_s	**2**	**5**	**10**	**15**	**20**
M-hidden ($p_1 = 0.63, p_2 = 0.32$)	0	0	0	0	0
M-hidden ($p_1 = 0.69, p_2 = 0.23$)	0	0	0	0	0
M-hidden ($p_1 = 0.75, p_2 = 0.14$)	0	0	0	0	0
M-hidden ($p_1 = 0.81, p_2 = 0.05$)	0	0	0	0	0
KHidden-M-rand	**100**%	**100**%	37%	0	0
KHidden-M-estimate	**100**%	**100**%	51%	0	0
KHidden-M-greedy	**100**%	**100**%	**88**%	**10**%	**1**%

Table 4: Success rates of the KHidden-M and M-hidden algorithms in generating instances that satisfy (2) when $maxH = 3n/4$.

$n_s = 20$ or $maxH = 3n/4$ and $n_s = 15$ or 20, the KHidden-M-rand algorithm cannot generate an SAT instance that satisfies the hardness condition in (2). For the KHidden-M-estimate algorithm, the success rate is also 0 when $maxH = 3n/4$ and $n_s = 15$ or 20. For the other parameter settings, the KHidden-M-rand, KHidden-M-estimate and KHidden-M-greedy algorithms have higher success rates than the M-hidden algorithm. The success rate of the KHidden-M-greedy algorithm remains higher than 0 for all of the settings of $maxH$ and n_s. These results show that the KHidden-M algorithm can be more effective than the M-hidden algorithm in generating SAT instances that satisfy the hardness condition in (2), and the KHidden-M algorithm is suitable for more general predefined solutions.

As seen in Tables 2, 3 and 4, the success rates of the KHidden-M-rand, KHidden-M-estimate and KHidden-M-greedy algorithms seem to decrease with $maxH$ and n_s. When $maxH$ and n_s are larger, it is more difficult for the three algorithms to generate SAT instances that satisfy the hardness condition in (2). The KHidden-M-estimate algorithm always obtains higher success rates than the KHidden-M-rand algorithm (except for parameter settings in which both algorithms have a success rate of 100%), and the KHidden-M-greedy algorithm achieves the highest success rates for all of the parameter settings. This indicates that the greedy strategy would be more effective than the other two strategies in generating SAT instances that satisfy the hardness condition in (2).

5.3.2 Hardness against WalkSAT

In this experiment, we investigate the performance of WalkSAT in solving the SAT instances that are generated by the KHidden-M algorithm and the M-hidden algorithm. We set n_s to 2, 5, 10, 15 and 20 and set $maxH$ to $n/4$, $n/2$, and $3n/4$. The other parameters are $n = 500$, $r = 5.6$, and $K = 3$. The predefined solutions are randomly generated according to the parameter settings. Next, the KHidden-M algorithm and the M-hidden algorithm are employed to generate SAT instances from the predefined solutions. After the SAT instances are generated, WalkSAT is employed to solve them regardless of whether those instances satisfy the hardness condition in (2). This experiment is conducted 29 times, and the number of flips made by WalkSAT is recorded to evaluate the hardness of the generated instances against WalkSAT.

n_s	**2**	**5**	**10**	**15**	**20**
M-hidden ($p_1 = 0.63, p_2 = 0.32$)	23208 (100%, -)	2430 (100%, -)	1321 (100%, -)	473 (100%, -)	585 (100%, -)
M-hidden ($p_1 = 0.69, p_2 = 0.23$)	75971 (100%, -)	3600 (100%, -)	1094 (100%, -)	757 (100%, -)	472 (100%, -)
M-hidden ($p_1 = 0.75, p_2 = 0.14$)	165813 (100%, -)	5150 (100%, -)	1067 (100%, -)	646 (100%, -)	539 (100%, -)
M-hidden ($p_1 = 0.81, p_2 = 0.05$)	**2938365** (86%,852231)	4551 (100%, -)	1117 (100%, -)	575 (100%, -)	382 (100%, -)
KHidden-M-rand	28344 (100%, -)	**5313** (100%, -)	1286 (100%, -)	938 (100%, -)	699 (100%, -)
KHidden-M-estimate	283 (100%, -)	374 (100%, -)	430 (100%, -)	878 (100%, -)	733 (100%, -)
KHidden-M-greedy	481 (100%, -)	3030 (89%,2576)	**31962** (100%, -)	**2417** (100%, -)	**855** (100%, -)

Table 5: Median number of flips made by WalkSAT in solving the instances generated by the KHidden-M and M-hidden algorithms when $maxH = n/4$. The parentheses include the percentage of successful runs and the median of successful runs; "-" indicates that it is consistent with the median number of flips.

n_s	**2**	**5**	**10**	**15**	**20**
M-hidden ($p_1 = 0.63, p_2 = 0.32$)	2721 (100%, -)	2988 (100%, -)	671 (100%, -)	473 (100%, -)	374 (100%, -)
M-hidden ($p_1 = 0.69, p_2 = 0.23$)	2974 (100%, -)	4412 (100%, -)	682 (100%, -)	424 (100%, -)	281 (100%, -)
M-hidden ($p_1 = 0.75, p_2 = 0.14$)	3010 (100%, -)	3341 (100%, -)	637 (100%, -)	408 (100%, -)	268 (100%, -)
M-hidden ($p_1 = 0.81, p_2 = 0.05$)	2893 (100%, -)	2369 (100%, -)	590 (100%, -)	351 (100%, -)	242 (100%, -)
KHidden-M-rand	147945 (100%, -)	**13403** (100%, -)	1394 (100%, -)	682 (100%, -)	403 (100%, -)
KHidden-M-estimate	459565 (100%, -)	2922 (100%, -)	1234 (100%, -)	805 (100%, -)	449 (100%, -)
KHidden-M-greedy	**100000000** (10%,18579)	4168 (100%, -)	**1502** (100%, -)	**1128** (100%, -)	**566** (100%, -)

Table 6: Median number of flips made by WalkSAT in solving the instances generated by the KHidden-M and M-hidden algorithms when $maxH = n/2$. The parentheses include the percentage of successful runs and the median of successful runs; "-" indicates that it is consistent with the median number of flips.

n_s	**2**	**5**	**10**	**15**	**20**
M-hidden ($p_1 = 0.63, p_2 = 0.32$)	2257 (100%, -)	4452 (100%, -)	849 (100%, -)	374 (100%, -)	274 (100%, -)
M-hidden ($p_1 = 0.69, p_2 = 0.23$)	2016 (100%, -)	6578 (100%, -)	791 (100%, -)	297 (100%, -)	269 (100%, -)
M-hidden ($p_1 = 0.75, p_2 = 0.14$)	2126 (100%, -)	4667 (100%, -)	572 (100%, -)	264 (100%, -)	255 (100%, -)
M-hidden ($p_1 = 0.81, p_2 = 0.05$)	1732 (100%, -)	4106 (100%, -)	607 (100%, -)	262 (100%, -)	233 (100%, -)
KHidden-M-rand	27043 (100%, -)	6998 (100%, -)	1601 (100%, -)	455 (100%, -)	259 (100%, -)
KHidden-M-estimate	6053 (100%, -)	**8818** (100%, -)	2078 (100%, -)	504 (100%, -)	296 (100%, -)
KHidden-M-greedy	**100000000** (0%,0)	7560 (100%, -)	**2574** (100%, -)	**754** (100%, -)	**510** (100%, -)

Table 7: Median number of flips made by WalkSAT in solving the instances generated by the KHidden-M and M-hidden algorithms when $maxH = 3n/4$. The parentheses include the percentage of successful runs and the median of successful runs; "-" indicates that it is consistent with the median number of flips.

The results for $maxH = n/4, n/2$ and $3n/4$ are shown in Tables 5, 6, and 7, respectively. The M-hidden algorithm with $p_1 = 0.81$ and $p_2 = 0.05$ achieves the best results when $maxH = n/4$ and $n_s = 2$. The reason might be that when $maxH$ and ns are small, the solutions for an SAT instance are located in a small region, and WalkSAT needs conduct more flips to find a solution, even if the SAT instance does not satisfy the hardness condition in (2). In this case, the randomness in generating the SAT instance might be more important, and the M-hidden algorithm contains more randomness than KHidden-M algorithm. For a similar reason, the KHidden-M-rand algorithm performs better than the KHidden-M-estimate and KHidden-M-greedy algorithms when $maxH = n/4$ and $n_s = 2$. As $maxH$ or n_s increases, the importance of satisfying the hardness condition in (2) increases. The KHidden-M algorithm performs better than the M-hidden algorithm for all of the other parameter settings. Specifically, the KHidden-M-rand algorithm achieves the best results when $maxH = n/4$ or $n/2$ and $n_s = 5$ (as shown in Tables 5 and 6). The KHidden-M-estimate algorithm achieves the best results when $maxH = 3n/4$ and $n_s = 5$ (as shown in Table 7). The KHidden-M-greedy algorithm achieves the best results at the remaining parameter settings (as shown in Tables 5, 6, and 7). Note that WalkSAT cannot solve the instances generated by the KHidden-M-greedy algorithm under the 10^8 flips limit when $maxH = n/2$ or $3n/4$ and $n_s = 2$. Overall, these results show that none of the three strategies for the KHidden-M algorithm can always perform better than the others, and the KHidden-M algorithm can perform better than the M-hidden algorithm when $maxH$ or n_s is large (e.g., $maxH \geq n/2$ or $n_s \geq 5$).

n_s	**2**	**5**	**10**	**15**	**20**
M-hidden ($p_1 = 0.63, p_2 = 0.32$)	0.385 (100%, -)	0.353 (100%, -)	0.352 (100%, -)	0.349 (100%, -)	0.341 (100%, -)
M-hidden ($p_1 = 0.69, p_2 = 0.23$)	1.036 (89%,0.997)	**0.357** (96%,0.356)	0.357 (100%, -)	0.350 (100%, -)	**0.344** (100%, -)
M-hidden ($p_1 = 0.75, p_2 = 0.14$)	66.665 (68%,1.015)	0.356 (100%, -)	0.352 (100%, -)	0.361 (100%, -)	0.268 (100%, -)
M-hidden ($p_1 = 0.81, p_2 = 0.05$)	**3596.230** (55%,377.879)	0.335 (100%, -)	0.327 (100%, -)	0.332 (100%, -)	0.138 (100%, -)
KHidden-M-rand	0.387 (100%, -)	0.353 (100%, -)	0.355 (100%, -)	0.360 (100%, -)	0.316 (100%, -)
KHidden-M-estimate	0.004 (100%, -)	0.015 (100%, -)	0.012 (100%, -)	0.070 (100%, -)	0.111 (100%, -)
KHidden-M-greedy	0.096 (100%, -)	0.354 (100%, -)	**40.551** (100%, -)	**0.378** (100%, -)	0.323 (100%, -)

Table 8: Median runtime (s) of Kissat in solving the instances generated by the KHidden-M and M-hidden algorithms when $maxH = n/4$. The parentheses include the percentage of successful runs and the median of successful runs; "-" indicates that it is consistent with the median runtime.

5.3.3 Hardness against Kissat

In this experiment, we investigate the performance of Kissat in solving the SAT instances generated by the KHidden-M algorithm and the M-hidden algorithm. We set n_s to 2, 5, 10, 15, and 20 and set $maxH$ to $n/4$, $n/2$, and $3n/4$. The other parameters are $n = 1000$, $r = 5.6$, and $K = 3$. The predefined solutions are randomly generated according to the parameter settings. Next, the KHidden-M algorithm and the M-hidden algorithm are employed to generate SAT instances from the predefined solutions. After the SAT instances are generated, Kissat is employed to solve them regardless of whether those instances satisfy the hardness condition in (2). This experiment is conducted 29 times, and the runtime of Kissat is recorded to evaluate the hardness of the generated instances against Kissat.

The results for $maxH = n/4, n/2$, and $3n/4$ are shown in Tables 8, 9, and 10, respectively. The hardness of the generated instances is roughly the same as the experimental results in the previous subsection. The M-hidden algorithm with $p_1 = 0.81$ and $p_2 = 0.05$ performs best when $maxH = n/4$ and $n_s = 2$, and Kissat only solves half of the instances. With the increase in $maxH$, the performance of the M-Hidden algorithm drops sharply, while the KHidden-M-greedy algorithm performs better. Even if more than half of the instances generated by the KHidden-M-geedy algorithm are successfully solved, the median runtime of successful runs is still approximately one hour.

When n_s not less than 5, most SAT instances generated by the M-hidden algorithm and the KHidden-M algorithm can be solved by Kissat within 1 second, and the gap between the results of different algorithms is small. Notably, the KHidden-M-greedy algorithm performs best when $maxH = n/4$ and $n_s = 10$, but Kissat only takes approximately 40 seconds

n_s	**2**	**5**	**10**	**15**	**20**
M-hidden ($p_1 = 0.63, p_2 = 0.32$)	0.351 (100%, -)	0.355 (100%, -)	0.345 (100%, -)	0.086 (100%, -)	0.021 (100%, -)
M-hidden ($p_1 = 0.69, p_2 = 0.23$)	0.366 (100%, -)	0.366 (100%, -)	0.349 (100%, -)	0.070 (100%, -)	0.013 (100%, -)
M-hidden ($p_1 = 0.75, p_2 = 0.14$)	0.344 (100%, -)	0.359 (100%, -)	0.329 (100%, -)	0.024 (100%, -)	0.012 (100%, -)
M-hidden ($p_1 = 0.81, p_2 = 0.05$)	0.331 (100%, -)	0.340 (100%, -)	0.236 (100%, -)	0.016 (100%, -)	0.005 (100%, -)
KHidden-M-rand	7.836 (93%,7.806)	0.346 (100%, -)	0.359 (100%, -)	0.330 (100%, -)	0.064 (100%, -)
KHidden-M-estimate	101.174 (93%,62.978)	**0.369** (100%, -)	**0.371** (100%, -)	**0.346** (100%, -)	0.106 (100%, -)
KHidden-M-greedy	**3566.569** (75%,3566.079)	0.365 (100%, -)	0.343 (100%, -)	0.322 (100%, -)	**0.242** (100%, -)

Table 9: Median runtime (s) of Kissat in solving the instances generated by the KHidden-M and M-hidden algorithms when $maxH = n/2$. The parentheses include the percentage of successful runs and the median of successful runs; "-" indicates that it is consistent with the median runtime.

n_s	**2**	**5**	**10**	**15**	**20**
M-hidden ($p_1 = 0.63, p_2 = 0.32$)	0.351 (100%, -)	0.357 (100%, -)	0.233 (100%, -)	0.022 (100%, -)	0.004 (100%, -)
M-hidden ($p_1 = 0.69, p_2 = 0.23$)	0.348 (100%, -)	0.365 (100%, -)	0.236 (100%, -)	0.023 (100%, -)	0.004 (100%, -)
M-hidden ($p_1 = 0.75, p_2 = 0.14$)	0.346 (100%, -)	**0.373** (100%, -)	0.288 (100%, -)	0.015 (100%, -)	0.004 (100%, -)
M-hidden ($p_1 = 0.81, p_2 = 0.05$)	0.329 (100%, -)	0.346 (100%, -)	0.099 (100%, -)	0.013 (100%, -)	0.003 (100%, -)
KHidden-M-rand	0.359 (100%, -)	0.347 (100%, -)	0.352 (100%, -)	0.092 (100%, -)	0.005 (100%, -)
KHidden-M-estimate	0.358 (100%, -)	0.360 (100%, -)	**0.390** (100%, -)	0.143 (100%, -)	0.005 (100%, -)
KHidden-M-greedy	**3569.578** (55%,3565.549)	0.362 (100%, -)	0.347 (100%, -)	**0.317** (100%, -)	**0.017** (100%, -)

Table 10: Median runtime (s) of Kissat in solving the instances generated by the KHidden-M and M-hidden algorithms when $maxH = 3n/4$. The parentheses include the percentage of successful runs and the median of successful runs; "-" indicates that it is consistent with the median runtime.

to solve the generated SAT instances. In fact, after our extensive experiments concluded that the SAT instances generated by our algorithms are intractable for Kissat only with the combination of specific parameter intervals: $r = 4.5$ for the KHidden-M-rand and KHidden-M-estimate algorithms, $r = 5.5$ for the KHidden-M-greedy algorithm, $n \geq 800$, $n/2 \leq maxH \leq 3n/4$ and $n_s \leq 2$.

6. Discussion

In this paper, we propose a more effective algorithm for generating hard SAT instances (with regard to the local search strategy) that hide multiple solutions. Experimental results show that compared with the M-hidden algorithm, our algorithm can generate harder SAT instances when the number of predefined solutions is large (e.g., not less than 10) or the Hamming distance between any two solutions is large (e.g., not less than half of the number of variables).

The proposed algorithm can be used for privacy preservation. Because each SAT instance can be converted to a negative database (*NDB*) and vice versa (Esponda et al., 2007; Esponda, 2008a), the KHidden-M algorithm can be used to generate hard-to-reverse NDBs that hide multiple strings, and thus, they can be used for privacy preservation. In some situations, the number of hidden strings is also critical and must be well protected, and this problem has been studied in previous works (Ateniese et al., 2011; Lindell et al., 2013; Bradley et al., 2016). Specifically, Ateniese (2011) described one scenario in which the U.S. Department of Homeland Security (DHS) attempts to check whether there is a suspected terrorist in an international flight that flies over the U.S. DHS has a list of suspected terrorists (denoted X), and the airline has a list of passengers (denoted Y) on the international flight. Obviously, the DHS wants to know the intersection of X and Y. If the airline does not belong to the U.S., it might be unwilling to directly provide the whole list of passengers to the DHS, and the DHS certainly would not directly provide the list of suspected terrorists to the airline. Moreover, the DHS does not want to reveal the size of its list (i.e., the number of suspected terrorists) to the airline because the size is also a state secret. A private set intersection protocol based on negative databases was proposed by Zhao et al. (2013), and it could be used to solve the above problem. In the protocol, a negative database is generated from X and Y to protect the concrete elements in X and Y and to hide the sizes of X and Y. Currently, several algorithms for generating an *NDB* with multiple hidden strings have been proposed, e.g., the prefix algorithm (Esponda et al., 2004, 2009), Randomized-*NDB* algorithm (Esponda et al., 2004, 2009) and M-hidden algorithm (Liu et al., 2015). However, the prefix algorithm cannot generate hard-to-reverse NDBs, and thus, the privacy of the hidden strings cannot be protected. The Randomized-*NDB* algorithm cannot ensure the hardness of reversing the generated *NDB*, and it is inefficient. Section 5 shows that the KHidden-M algorithm can be more effective than the M-hidden algorithm. Therefore, it is reasonable to employ the KHidden-M algorithm in the proposed protocol (Zhao & Luo, 2013) to solve the above problem. Luo et al. (2018) summarized more applications of NDBs, e.g., authentication, secure multiparty computation, information hiding, privacy-preserving data mining and data publication.

Another advantage of using the KHidden-M algorithm to hide multiple strings/solutions is that some extra strings can be hidden together to further enhance the security of the

predefined hidden strings. To the best of our knowledge, most existing SAT solvers are designed with such a mechanism: if a feasible solution is found, then the solver will return the solution, and it will not continue to find other solutions. Finding all strings hidden by NDBs can be much harder than finding only one hidden string. When generating an *NDB* using the KHidden-M algorithm, the hardness conditions with regard to extra hidden strings need not be satisfied. In this case, SAT solvers can find an extra hidden string as the solution more easily, and thus, the security of the predefined solutions can be enhanced by the cover of extra hidden solutions. Moreover, even if attackers have reversed the NDBs and have found all of the hidden strings, they cannot determine which strings are the secrets.

There are also some issues with the KHidden-M algorithm that might need to be further studied. For example, we consider only the hardness of the SAT instances against the local search strategy when designing the KHidden-M algorithm, and the hardness of the KHidden-M instances against other strategies that are used in state-of-the-art SAT solvers also needs to be investigated in future work. Some prior works have demonstrated a strong correlation between the performance of CDCL solvers and the community structure of SAT instances (Ansótegui et al., 2009; Giráldez-Cru & Levy, 2016). In fact, we have already conducted some preliminary follow up on this work, but simply dividing the variables into communities and then generating intracommunity/intercommunity clauses in our model did not lead to better results. Our model appears to attenuate the effects of community structure in the generated SAT instances and vice versa. How to better integrate our model with the community structure is an expected research direction in future work. In addition, it is worthwhile to further consider the scale-free structure of SAT instances in our model. We will also try to use our model to generate appropriate SAT instances and submit them to future SAT Competition/Race/Challenge as benchmarks.

7. Conclusion

In this paper, we propose the KHidden-M algorithm to generate hard SAT instances (against the local search strategy) that hide multiple predefined solutions. Three strategies for generating clauses that are satisfied by all predefined solutions are proposed, and each can be employed in the KHidden-M algorithm. If the generated clauses do not satisfy the hardness condition for the local search strategy, the KHidden-M algorithm will call an adjustment strategy to tune the clauses. The experimental results prove the effectiveness of the KHidden-M algorithm. Overall, the KHidden-M-greedy algorithm outperforms the other two algorithms in most parameter settings, but it also exhibits more significant instability than the other two algorithms. When the number of variables is large (e.g., greater than 800), we may prefer the KHidden-M-estimate algorithm. When the number of predefined solutions and the Hamming distance between them are small, we may prefer the KHidden-M-rand algorithm. In other cases, we may prefer the KHidden-M-greedy algorithm. Each of the three algorithms has applicable scenarios. Moreover, the KHidden-M algorithm can generate harder SAT instances (against WalkSAT) than the M-hidden algorithm. The experimental results also show that with specific parameter settings, the KHidden-M instance is intractable for Kissat. The proposed algorithm can be used to generate benchmarks for the SAT Competition/Race/Challenge. The KHidden-M algorithm can also be used in some applications, e.g., secure iris recognition and secure multiparty computation.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (Grant No. 61806151, 61175045, 61672398, 62006001) and the Natural Science Foundation of Chongqing City (Grant No. cstc2021jcyj-msxmX0002).

References

Achlioptas, D. (2001). Lower bounds for random 3-SAT via differential equations. *Theoretical Computer Science*, *265*(1-2), 159–185.

Achlioptas, D., Gomes, C., Kautz, H., & Selman, B. (2000). Generating satisfiable problem instances. *AAAI/IAAI*, *2000*, 256–261.

Achlioptas, D., Jia, H., & Moore, C. (2005). Hiding satisfying assignments: Two are better than one. *Journal of Artificial Intelligence Research*, *25*, 623–639.

Ansótegui, C., Bonet, M. L., & Levy, J. (2009). Towards industrial-like random sat instances. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, p. 387–392.

Ateniese, G., Cristofaro, E. D., & Tsudik, G. (2011). (if) size matters: size-hiding private set intersection. In *International Workshop on Public Key Cryptography*, pp. 156–173.

Barak-Pelleg, D., Berend, D., & Saunders, J. (2022). A model of random industrial sat. *Theoretical Computer Science*, *910*, 91–112.

Barthel, W., Hartmann, A. K., Leone, M., Ricci-Tersenghi, F., Weigt, M., & Zecchina, R. (2002). Hiding solutions in random satisfiability problems: A statistical mechanics approach. *Physical Review Letters*, *88*(18), 188701.

Boggs, P. T., & Tolle, J. W. (1995). Sequential quadratic programming. *Acta numerica*, *4*, 1–51.

Bradley, T., Faber, S., & Tsudik, G. (2016). Bounded size-hiding private set intersection. In *International Conference on Security and Cryptography for Networks*, pp. 449–467.

Bringer, J., & Chabanne, H. (2010). Negative databases for biometric data. In *Proceedings of the 12th ACM Workshop on Multimedia and Security*, pp. 55–62.

Dasgupta, D., & Azeem, R. (2008). An investigation of negative authentication systems. In *Proceedings of 3rd international conference on information warfare and security*, pp. 117–126.

Dasgupta, D., & Saha, S. (2009). A biologically inspired password authentication system. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, pp. 41–44.

Esponda, F. (2008a). Everything that is not important: Negative databases. *IEEE Computational Intelligence Magazine*, *3*(2), 60–63.

Esponda, F. (2008b). Hiding a needle in a haystack using negative databases. In *International Workshop on Information Hiding*, pp. 15–29.

Esponda, F., Ackley, E. S., Forrest, S., & Helman, P. (2004). Online negative database. In *Proceedings of the Third International Conference on Artificial Immune Systems*, pp. 175–188.

Esponda, F., Ackley, E. S., Helman, P., Jia, H., & Forrest, S. (2007). Protecting data privacy through hard-to-reverse negative databases. *International Journal of Information Security*, *6*(6), 403–415.

Esponda, F., Forrest, S., & Helman, P. (2004). Enhancing privacy through negative representations of data. *University of New Mexico, TR-CS-2004-18*, *50*, 193.

Esponda, F., Forrest, S., & Helman, P. (2009). Negative representations of information. *International Journal of Information Security*, *8*(5), 331–345.

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

Giráldez-Cru, J., & Levy, J. (2016). Generating sat instances with community structure. *Artificial Intelligence*, *238*, 119–134.

Giráldez-Cru, J., & Levy, J. (2021). Popularity-similarity random sat formulas. *Artificial Intelligence*, *299*, 103537.

Giráldez-Cru, J., & Levy, J. (2017). Locality in random sat instances. In *International Joint Conferences on Artificial Intelligence*, p. 638–644.

Jia, H., Moore, C., & Strain, D. (2007). Generating hard satisfiable formulas by hiding solutions deceptively. *Journal of Artificial Intelligence Research*, *28*, 107–118.

Lindell, Y., Nissim, K., & Orlandi, C. (2013). Hiding the input-size in secure two-party computation. In *International Conference on the Theory and Application of Cryptology and Information Security 2013*, pp. 421–440.

Liu, R., Luo, W., & Yue, L. (2014). The p-hidden algorithm: Hiding single databases more deeply. *International Journal of Immune Computation*, *2*(1), 43–55.

Liu, R., Luo, W., & Yue, L. (2015). Hiding multiple solutions in a hard 3-SAT formula. *Data & Knowledge Engineering*, *100, Part A*, 1–18.

Luo, W., Hu, Y., Jiang, H., & Wang, J. (2019). Authentication by encrypted negative password. *IEEE Transactions on Information Forensics and Security*, *In press, DOI: 10.1109/TIFS.2018.2844854*.

Luo, W., Liu, R., Jiang, H., Zhao, D., & Wu, L. (2018). Three branches of negative representation of information: A survey. *IEEE Transactions on Emerging Topics in Computational Intelligence*, *2*(6), 411–425.

Omelchenko, O., & Bulatov, A. (2021). Satisfiability and algorithms for non-uniform random k-SAT. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35, pp. 3886–3894.

SAT Competition (2016). Call for benchmarks. URL: https://baldur. iti.kit.edu/sat-competition-2016/index.php?cat=benchmarks..

Selman, B., Kautz, H. A., Cohen, B., et al. (1995). Local search strategies for satisfiability testing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, *26*, 521–532.

Selman, B., Mitchell, D. G., & Levesque, H. J. (1996). Generating hard satisfiability problems. *Artificial Intelligence*, *81*(1-2), 17–29.

Xu, K., Boussemart, F., Hemery, F., & Lecoutre, C. (2007). Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artificial Intelligence*, *171*(8-9), 514–534.

Xu, K., & Li, W. (2000). Exact phase transitions in random constraint satisfaction problems. *Journal of Artificial Intelligence Research*, *12*, 93–103.

Xu, K., & Li, W. (2006). Many hard examples in exact phase transitions. *Theoretical Computer Science*, *355*(3), 291–302.

Zhao, D., & Luo, W. (2013). A study of the private set intersection protocol based on negative databases. In *Proceedings of the 11th IEEE International Conference on Dependable, Autonomic and Secure Computing*, pp. 58–64.

Zhao, D., Luo, W., Liu, R., & Yue, L. (2015a). A fine-grained algorithm for generating hard-to-reverse negative databases. In *Proceedings of the 2015 International Workshop on Artificial Immune Systems*, pp. 1–8.

Zhao, D., Luo, W., Liu, R., & Yue, L. (2015b). Negative iris recognition. *IEEE Transactions on Dependable and Secure Computing*, *15*(1), 112–125.

Zhao, D., Luo, W., Liu, R., & Yue, L. (2017). Experimental analyses of the K-hidden algorithm. *Engineering Applications of Artificial Intelligence*, *62*, 331–340.