algorithms

MDPI

Article

An Experimental Survey of Extended Resolution Effects for SAT Solvers on the Pigeonhole Principle

Tomohiro Sonobe

National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan; tomohiro_sonobe@nii.ac.jp

Abstract: It has been proven that extended resolution (ER) has more powerful reasoning than general resolution for the pigeonhole principle in Cook's paper. This fact indicates the possibility that a solver based on extended resolution can exceed Boolean satisfiability problem solvers (SAT solvers for short) based on general resolution. However, few studies have provided practical evidence of this assumption. This paper explores how extended resolution can improve SAT solvers by using the pigeonhole principle, which was the first problem solved by ER in polynomial steps. In fact, although Cook's paper introduced how to add auxiliary variables, there is no evidence that these variables are really useful for practical solvers. We try to answer the question: If the SAT solver can add appropriate auxiliary variables as proposed in Cook's paper, can the solver enhance its performance by utilizing these variables? Experimental results show that if the solver properly prioritizes the extended variables in the search, the solver can end the search in a shorter time.

Keywords: search; SAT solver; extended resolution

Citation: Sonobe, T. An Experimental Survey of Extended Resolution Effects for SAT Solvers on the Pigeonhole Principle. *Algorithms* **2022**, *15*, 479. https://doi.org/10.3390/a15120479

Academic Editor: Hirotaka Ono

Received: 17 November 2022
Accepted: 14 December 2022
Published: 16 December 2022

1. Introduction

The Boolean satisfiability problem (SAT problem for short) deals with whether specific Boolean variables can evaluate whether a given Boolean formula is true. The SAT problem is known as the first NP-complete problem [1]. Many practical problems can be reduced to SAT, and there are many applications, such as circuit design [2], neural network verification [3], and mathematical problem solving [4–6], where SAT can be applied.

To solve the Boolean satisifiability problem, we use what is called a SAT solver which is based on general resolution. The state-of-the-art SAT solvers are based on a backtrack search algorithm, called the Davis–Putnum–Logemann–Loveland (DPLL) algorithm [7]. The DPLL has been enhanced by a conflict clause learning mechanism [8], and today the DPLL, combined with clause learning, is called the Conflict-Driven Clause Learning (CDCL) algorithm. Clause learning conducts multiple resolution processes when a conflict occurs in a search and then derives a new learned clause. On the other hand, extended resolution (ER) [9] is different from general resolution. ER introduces new variables (i.e., extended variables) that do not exist in a given formula. Cook proved that the pigeonhole principle can be solved in polynomial steps by using extended resolution [10]. From this point of view, there is a possibility that solvers with ER can outperform the solvers based on CDCL, at least for the pigeonhole principle.

However, today's SAT solvers are still based on CDCL that uses general resolution. One of the main reasons is that there is no stable way to choose effective extended variables. Although existing research such as [11,12] tried to find extended variables that reduce the size of clauses (or resolution steps), there is much room for improvement. In this paper, we try to answer the question: If the SAT solver can add appropriate extended variables as proposed in [10], can the solver enhance its performance by utilizing these variables for the pigeonhole principle? The extended variables introduced in [10] are theoretically the most suitable but their practical application is not clear. As a first step, we confirm that

trying to solve a SAT problem by adding extended variables has almost no effect. Then, we analyze two simple methods to leverage these variables in a search. From experimental results, we show that our method can speed up the overall processing time. In addition, analytical results of the DRUP (Delete Reverse Unit Propagation) proof [13] reveal that our method can utilize extended variables more frequently. We provide practical evidence that extended resolution (variables) can boost the performance of CDCL solvers.

Our contributions are as follows:

- We introduce two ways to add extended variables: full extension introduced by [10] and partial extension by restricting a large part of the variables in [10].
- We show that trying to solve instances of the pigeonhole principle with full and partial extension using basic solvers (MiniSAT [14] and Glucose [15]) has almost no effect.
- We analyze two simple methods to utilize extended variables in the search. Experimental results indicate that these methods can improve the performance of the solvers. This fact implies that if extended variables are properly selected and properly utilized, the extended resolution can practically improve the performance of the pigeonhole principle.

Note that our aim is neither to tackle difficult (or large) pigeonhole principle instances (e.g., [16]) nor to develop a specific solver better than state-of-the-art solvers for the pigeonhole principle, but to analyze the effect of extended resolution based on [10]. From that perspective, the implementation of our contribution is a proof of concept. However, our proposed methods can be easily applied to state-of-the-art solvers because the methods do not depend on implementation of specific solvers. Although we assume the ideal situation that how to add extended variables is known in advance, our experimental results can be a guide to build ER solvers.

The rest of the paper is organized as follows. Section 2 explains the background of this paper, and Section 3 introduces some existing works. Section 4 details procedures of our analysis, and Section 5 shows experimental results. Finally, Section 6 concludes the paper.

2. Preliminaries

2.1. SAT Problem

The SAT problem consists of Boolean variables. A literal is either a positive form or a negative form of a Boolean variable. A positive (negative) literal is classified as true if the corresponding variable is deemed to be true (false). The SAT problem is given as Conjunctive Normal Form (CNF) in general, where a CNF consists of a conjunction (logical and) of clauses, and a clause consists of a disjunction (logical or) of literals. The SAT problem asks whether a variable assignment that satisfies (i.e., evaluates as true) the given CNF exists. Cook proved the SAT problem is the first NP-complete problem [1]. An SAT instance is satisfiable if there is at least one valid variable assignment, otherwise it is unsatisfiable.

2.2. SAT Solver

Solvers for SAT problem are called SAT solvers. The mainstream algorithm for practical instances (no random generation) is a backtracking search, which is called the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [7]. The DPLL has been enhanced by adding a conflict clause learning mechanism [8], and today the DPLL combined with the clause learning is called the Conflict-Driven Clause Learning (CDCL) algorithm. Algorithm 1 shows the pseudo code of the CDCL. The details are in [14].

In the algorithm, unit propagation Line 9) conducts forcing assignments to variables in unit clauses in which only one remaining literal is unassigned, and the others are false. After propagation, there can be a conflicting clause. A clause is in conflict when all of its literals are false. The clause learning (Line 11) conducts multiple resolution processes

when a conflict in the search occurs (Line 10). Suppose there are two clauses, $(x \vee A)$ and $(\neg x \vee B)$, where A and B are sub-clauses. Resolution on the variable x is defined as follows:

$$\frac{(x \vee A), (\neg x \vee B)}{(A \vee B)}$$

This is the reason why the CDCL solvers are based on the resolution. The most popular learning scheme (i.e., what kind of clauses should be selected for resolution) is first-UIP [17].

Algorithm 1 Pseudo code of CDCL.

```
Input: a CNF formula ∏
Output: SATISFIABLE or UNSATISFIABLE
 1: level = 0 // decision level (depth of search tree)
 2: learnts = ∅ // learnt clauses
 3: inc_score = 1.0 // incremental value for VSIDS scores
 4: conf // conflicting clause
 5: learnt // learnt clause
 6: blevel // level to backtrack
 7: next // next decision variable
 8: while true do
 9:   conf = unitPropagation(∏)
10:   if conf ≠ NULL then
11:     learnt = conflictAnalysis(∏, conf)
12:     blevel = calcBackjumpLevel(learnt)
13:     if blevel < 0 then
14:       return UNSATISFIABLE
15:     end if
16:     for each var in learnt do
17:       increaseVSIDSScores(var, inc_score)
18:     end for
19:     inc_score = inc_score / 0.95
20:     learnts = learnts ∪ learnt
21:     if restart() then
22:       blevel = 0
23:     end if
24:     backjump(blevel)
25:     level = blevel
26:   else
27:     next = chooseDecisionVariable(∏)
28:     if next == NULL then
29:       return SATISFIABLE
30:     end if
31:     assignValue(next)
32:     level = level + 1
33:   end if
34: end while
```

After learning a learnt clause, backjump (Line 24) [18] undoes the variable assignment to a certain level (depth of search tree) where the newly learnt clause becomes a unit. The level to backjump is calculated at Line 12 ("calcBackjumpLevel" function) from the learnt clause. If the learnt clause cannot be a unit, then the given formula is unsatisfiable (Lines 13 and 14). While general backtracking just cancels the assignment of the last variable, backjumping directly goes back to the culprit variable that causes the conflict (and is included in the learnt clause).

Restart (Line 21) [19] cancels all the variable assignments. SAT solvers are often stuck in a desert search space where no solution exists and no useful information (i.e., useful

learnt clause) can be extracted. Restart is effective to escape such a search area. There are two types for determining when to restart: statically [20] and dynamically [21].

If there is no conflict after the unit propagation, decision process chooses a variable, called decision variable, to be assigned (Line 27). The most popular decision heuristic is the Variable State Independent Decaying Sum (VSIDS) [17]. The VSIDS attaches a score for each variable and chooses the highest scored variable at decision. The scores are increased when the corresponding variable is included in a learnt clause (Lines 16 and 17). The incremental value for the VSIDS score grows exponentially (Line 19) to prioritize variables recently assigned and to neglect past scores. In recent solvers, machine-learning-based scores such as Learning Rate Branching (LRB) [22] are used together. There are also other decision techniques that combine multiple methods [23,24]. If all the variables are assigned, then the given formula is satisfiable (Lines 28 and 29).

Before starting the search, preprocessing [25–27] simplifies the given CNF. Instances made from real-world applications sometimes include redundant clauses. For example, a clause $A = (a \vee b \vee c)$ is redundant if there is a clause $B = (a \vee b)$ because when B is satisfied A is also satisfied. Subsumption, as one of preprocessing techniques, removes these clauses. Recent solvers conduct inprocessing [28] that simplifies the clauses and variables by using techniques of preprocessing during a search.

2.3. Pigeonhole Principle

The pigeonhole principle (PHP) [29] is defined as follows.

Definition 1. *(Pigeonhole Principle). The pigeonhole principle asks whether all $n > 1$ pigeons can enter $n - 1$ holes so that one pigeon is at one hole. In other word, there is no injective function mapping from $\{1, 2, \ldots, n\}$ to $\{1, 2, \ldots, n-1\}$ for $n > 1$.*

The answer is clearly no. Thus a SAT instance converted from the pigeonhole principle is unsatisfiable. For each integer $n > 1$, the pigeonhole principle is formulated as follows. A variable P_{ij} for $1 \leq i \leq n$ and $i \leq j \leq n-1$ is true when pigeon i is at hole j. The constraints are described as a set of clauses: one pigeon at least in one hole (at least one constraint) is

$$(P_{i,1} \vee P_{i,2} \vee \ldots \vee P_{i,n-1}) \; for \; 1 \leq i \leq n \tag{1}$$

and one hole containing at most one pigeon (at most one constraint) is

$$(\neg P_{i,k} \vee \neg P_{j,k}) \; for \; 1 \leq i < j \leq n, 1 \leq k \leq n-1 \tag{2}$$

The pigeonhole principle formula S_n constitutes the conjunction of Formulas (1) and (2).

Cook [10] proved that the PHP can be solved in polynomial steps by using extended resolution [9]. Given a formula F, extended resolution is a proof system to apply the general resolution on the extended formula $F*$, where $F*$ is obtained by adding redundant clauses to F. Cook introduced a new variable $Q_{i,j}$ below:

$$Q_{i,j} \equiv (P_{i,j} \vee (P_{i,n-1} \wedge P_{n,j})) \; for \; 1 \leq i \leq n-1, \; 1 \leq j \leq n-2 \tag{3}$$

The formula is expressed as the following four clauses:

$$Q_{i,j} \vee \neg P_{i,j}$$

$$Q_{i,j} \vee \neg P_{i,n-1} \vee \neg P_{n,j}$$

$$\neg Q_{i,j} \vee P_{i,j} \vee P_{i,n-1}$$

$$\neg Q_{i,j} \vee P_{i,j} \vee P_{n,j}$$

The addition of these clauses to the original PHP formula guarantees the satisfiability equivalency. We describe the newly introduced variable $Q_{i,j}$ as an extended variable and a clause including extended variables as an extended clause. Using $O(n^3)$ resolutions

on every $P_{i,j}$ variable, only the extended variables remain. Hence, the formula S_n can be reduced to S_{n-1}. By repeating the same manner, we can derive $S_2 = P_{1,1} \wedge P_{2,1} \wedge (\neg P_{1,1} \vee \neg P_{2,1})$ that can be easily proved to be unsatisfiable.

Example 1. *For a case where the number of pigeons is 3 ($n = 3$), Formula (1) is*
$(P_{1,1} \vee P_{1,2}) \wedge (P_{2,1} \vee P_{2,2}) \wedge (P_{3,1} \vee P_{3,2})$
Formula (2) is
$(\neg P_{1,1} \vee \neg P_{2,1}) \wedge (\neg P_{1,1} \vee \neg P_{3,1}) \wedge (\neg P_{2,1} \vee \neg P_{3,1}) \wedge (\neg P_{1,2} \vee \neg P_{2,2}) \wedge (\neg P_{1,2} \vee \neg P_{3,2}) \wedge (\neg P_{2,2} \vee \neg P_{3,2})$
Formula (3) is
$Q_{1,1} \equiv (P_{1,1} \vee (P_{1,2} \wedge P_{3,1})) \wedge Q_{2,1} \equiv (P_{2,1} \vee (P_{2,2} \wedge P_{3,1}))$

3. Related Work

There are various works related to extended resolution for SAT solvers.

Tseitin [9] first introduced the extension rule adding a new variable to the original formula and retaining its satisfiability-equivalency. Cook [10] used this extension rule, i.e., as extended resolution, and established the polynomial-step solution of the pigeonhole principle. Although Tseitin's original extension was the form $x \leftrightarrow (\neg a \vee \neg b)$, more general extensions (including [10]) are commonly used [30]. Kullmann [31] generalized extended resolution as an addition of blocked clauses and proved it can speed up resolution proofs (later, in reverse, blocked clause elimination [32] was shown to be helpful for SAT solvers). DRAT (Deletion Resolution Asymmetric Tautology) [33], used as the standard proof format (known as the successor of DRUP) for unsatisfiable instances, has also been proven to generalize extended resolution [34,35].

Huang [11] proposed extended clause learning (ECL). It conducts an extended resolution when a conflict occurs. When a clause $\gamma = \alpha \vee \beta$ such that $|\alpha| \geq 2$ and $|\beta| \geq 1$ is learnt by a standard learning scheme (e.g., 1-UIP), the ECL learns clauses $(x \vee \beta)$ and $(x \leftrightarrow \alpha)$ instead of γ where x is a fresh variable. The ECL only applies the extension when $|\gamma| > 30$ and conducts a restart after the extension. The author implemented the ECL on Tinisat [36] and confirmed that TiniSAT with ECL could solve more instances for specific problems. Audemard et al. [12] proposed a local extension. When two consecutive learnt clauses $C_i = (\neg l_1 \vee D)$ and $C_{i+1} = (\neg l_2 \vee D)$ are derived, a new variable $v \leftrightarrow (l_1 \vee l_2)$ is introduced. The clauses C_i and C_{i+1} can be replaced with $C' = (\neg v \vee D)$; thus, the number of learnt clauses can be reduced. Moreover , all clauses, including l_1 and l_2, can be replaced by using the variable v (a clause $(l_1 \vee l_2 \vee A)$ is equivalent to $(v \vee A)$). They implemented the proposed method on MiniSAT 2.2 and (the first version of) Glucose and confirmed performance improvements on instances of SAT Competition 2007 and 2009. Manthey [37] improved the work by using a Bloom filter and reducing some overheads. Jabbour et al. [38] proposed a way to detect auxiliary variables by exploiting the hidden Boolean functions used for the problem encoding. The method tries to detect a new variable $y = (x_1 \vee \ldots \vee x_n)$ or $y = (x_1 \wedge \ldots \wedge x_n)$ (equivalent to $\neg y = (\neg x_1 \vee \ldots \vee \neg x_n)$) during the search and then substitutes the variable y for clauses including $(x_1 \vee \ldots \vee x_n)$. They implemented the method to MiniSAT 2.2 and attained a performance gain on specific problem instances. Manthey proposed blocked variable addition (BVA) [39] as a preprocessing technique. BVA introduces a new variable that reduces the number of clauses in a given CNF instance, in reverse manner of general resolution. Although we tried BVA implemented in a coprocessor [40] on PHP instances as a preliminary experiment, BVA did not reproduce the extended variables introduced by Cook [10]. Ignatiev et al. [16] tackled the PHP as a MAXSAT problem by transforming the original SAT problem. In fact, they solved larger instances than the ones we try. Our focus is not to challenge solving difficult (or large) instances but to analyze the practical effect of extended resolution in a (theoretically) ideal condition.

As a representative instance that extended resolution was proved to be powerful in theory, we show the practical usefulness of extended resolution on the PHP in this paper.

4. Analysis Procedure

We analyze the effect of extended resolution on the PHP, i.e., the effect of extended variables, by asking two questions: (1) Can extended variables speed up a solver's search? and (2) How do extended variables affect proof of a formula's unsatisfiability? For speeding up solvers, we introduce two ways to prioritize the extended variables as decision variables. Then, we observe whether the extended variables can really enhance the search efficiency by measuring processing times. Moreover, we also survey the solver DRUP's proof [13] that represents the resolution steps leading to an empty clause, i.e., unsatisfiability of the formula.

4.1. Full and Partial Extension

We use two types of variable extension, full and partial. Full extension is the same as the one proposed in [10], using all the extended variable $Q_{i,j}$ in Formula (3). Partial extension uses $Q_{i,j}$ for only $i = 1$. The right-hand side of Formula (3), $P_{i,j} \vee (P_{i,n-1} \wedge P_{n,j})$, is equivalent to $(\neg P_{i,j} \Rightarrow (P_{i,n-1} \wedge P_{n,j}))$, which propositionally means that if pigeon i does not enter into hole j, then pigeon i enters into hole $n-1$ and pigeon n enters into hole j. Thus, if $P_{i,j} = False$, then pigeon i and n are fixed to hole $n-1$ and j, respectively. This constraint can lead to a shortcut because positions of two pigeons are fixed by assigning only one variable ($P_{i,j}$). Note that all the pigeons are not forced to enter a specific hole in the original formula, which means that proving the unsatisfiability needs to consider all the permutation of pigeon-hole assignments. Therefore, adding extended variables only for a pigeon ($i = 1$) can be sufficiently helpful to SAT solvers.

4.2. Prioritizing Extended Variables

The extended variables are actually redundant from the perspective of logical equivalence; the satisfiability of a formula without extended variables does not change. In fact, the problem size expansion can deteriorate the solver's performance, and it is the case for extended variables (see the experimental results below). To reduce the redundancy of a SAT problem, state-of-the-art SAT solvers conduct preprocessing on the given instance. Preprocessing [26] of a CNF removes some variables and clauses to shrink the size of the problem instance while preserving the logical equivalence. Preprocessing comprises subsumption, variable elimination, and hyper resolution. Extended variables are also the target of preprocessing (i.e., variable elimination). Hence, we switch off the preprocessing in the experiments.

Merely adding extended variables in the original formula does not affect the solvers since they do not know which variables are extended ones or not. The solvers should utilize these variables to speed up their efficiency. To this end, we introduce two methods to prioritize the extended variables as decision variables at the early phase of the search. The actual codes are in Appendix A.

One is a simple way: extended variables are always selected prior to non-extended variables. For each decision, the solver chooses an unassigned extended variable with the highest VSIDS score. If there remain no unassigned extended variables, then a non-extended variable is selected. We call this method "m1". In practice, the method is executed in the "chooseDecisionVariable" function in Algorithm 1.

The other way is shown in Algorithm 2. We denote this method "m2". This function is called after every restart, just after the "restart()" function in Algorithm 1. The constant R is greater than 1 in order to add larger VSIDS score than the usual (inc_score is the regular value for increasing VSIDS scores). It induces the extended variables being selected as decision variables at the early phase after a restart. We try a couple of R values in the following experiments and choose the best one.

Algorithm 2 Pseudo code of "m2" method.

```
inc_score // incremental value for VSIDS scores
for each extended variable ev do
    increaseVSIDSScores(ev, inc_score * R)
end for
```

4.3. UNSAT Core Analysis

In addition to processing time, we also analyze the DRUP proof for each instance. The DRUP proof is a record of how the empty clause (causing the unsatisfiability of the instance) is derived: which clause is used for resolution to generate a new learnt clause. By analyzing this core, we can count the frequency of each clause that contributes to prove the unsatisfiability. For example, assume that there is an unsatisfiable CNF that consists of three clauses, A, B, and C. A SAT solver outputs a DRUP proof that tells that a learnt clause D is generated from A and B, a learnt clause E is from B and C, and finally an empty clause is from D and E. In this case, the frequency of each clause, A, B, and C, is 1, 2, and 1, respectively (note that clause B is used twice for deriving clause D and E).

If extended variables and its clauses really impact the solution, these clauses can appear frequently in the DRUP proof. We use DRUP-trim [13] to translate the DRUP proof into a resolution graph (TRACECHECK format (http://fmv.jku.at/tracecheck/README.tracecheck, accessed on 20 November 2022)), and then count the frequency of each clause. After sorting the frequency, we observe the frequently used clauses (e.g., top-100).

The TRACECHECK format is as follows:

$$c_{index},\ l_1,\ l_2,\ \ldots,\ l_x,\ 0,\ a_1,\ a_2,\ \ldots,\ a_y,\ 0$$

Each element is separated by a single space (commas are used for readability in the above example). One line corresponds to one clause. The number 0 is used as a delimiter. The c_{index} stands for its index: original clauses in a CNF with m clauses have an index 1 to m, and derived (learnt) clauses have an index more than m. The clause has x literals: $l_1, l_2, \ldots, l_x$. If the clause is a derived one, the clauses with index a_1 to a_y are the antecedents that are used to derive the clause by resolution (note that all antecedent indices are less than the c_{index}).

We denote D_i as a set of descendants of a clause c_i, and define the frequency of a clause $freq[c_i]$ for a DRUP by traversing clause indices in reverse order as follows:

$$freq[c_i] = \sum_{c_j \in D_i} freq[c_j] \tag{4}$$

Note that the frequency of a clause without any descendants is 1. Since the frequency increases exponentially, it is difficult to calculate exact values with the standard 32- or 64-bit integer used in the standard programming languages. We instead calculate approximated values with 64-bit double floating-point numbers: when the highest frequency exceeds a threshold (we set it as 10^{200}) we multiply all the frequencies with 10^{-200}. This approximation can retain the magnitude relationship of the frequencies. Even though some small frequency values can be rounded to 0 due to underflow, we ignore these infrequent ones. In this manner, we finally examine the relative values in descending order, i.e., ranking of frequencies of all the clauses in which we can see the frequently used clauses in a resolution proof.

5. Experimental Results

We conduct computational experiments to see how extended variables affect and improve the search efficiency. First, we see the performance (processing time) of solvers with/without one of the methods shown in Section 4.2 on three types of instances (no extension, full extension, and partial extension) in Section 4.1. Second, we analyze DRUP proofs generated from solvers on the three extensions and observe how many times extended

variables are used to derive unsatisfiability. For both experiments, we use two popular SAT solvers: MiniSAT 2.2 [14] and Glucose 3 [15]. Both solvers are the most fundamental solvers used as a basis for today's subsequent solvers. We switch off the preprocessing, as described in Section 4.2, so not to remove the extended variables. In preliminary experiments, we found that the preprocessing for instances of no extension deteriorates the solvers' performance. Hence, we report results without the preprocessing even for (not extended) original PHP instances.

We use the PHP instances with the number of pigeons (n) from 11 to 15. Note that instances with less than 10 pigeons are easy to solve within 10 seconds in our preliminary experience. We set the time limit for each instance as 10,000 s and the constant R in Algorithm 2 as 100 that showed good performance among the values 10, 100, 1000, and 10,000 in our preliminary experiments. Since the order of clauses and variables in a CNF can affect the solver's performance [41] (e.g., initial decision variables and initial watched literals [14]), we shuffle the order by random reordering and make four differently shuffled instances for each n.

We use a machine running Ubuntu 16.04 with two Xeon Gold 5218 CPUs and 384 GB RAM. Our codes, MiniSAT 2.2 and Glucose 3 with/without our proposed method, are compiled by GNU C++ Compiler version 5.4.0. Since both solvers run deterministically, we execute each solver on each instance once.

5.1. Performance Comparison

Table 1 exhibits the results of vanilla (no extension) PHP instances. The second to sixth columns (n, #V, #C, #EV, and #EC) indicate the number of pigeons, variables (including extended variables), clauses (including extended clauses), extended variables, and clauses including extended variables, respectively. The seventh and later columns stand for the results of each solver. The "M" and "G" indicate MiniSAT 2.2 and Glucose 3, respectively. A solver with one of our methods is expressed as "M/G +m1/m2". The values for each solver indicate the processing time in seconds, and the best one is bolded. Instances with more pigeons are difficult to solve from the point of view of processing time. Specifically, instances with more than 13 pigeons could not be solved by any solvers. Since Glucose is the enhanced solver based on MiniSAT, it could solve instances with 13 pigeons. Note that due to the lack of extended variables, our two methods have essentially no effect for these instances.

Table 2 exhibits the results of the PHP instances with full extension. Glucose with "m1" and Glucose with "m2" exhibited good performance, especially for large instances. Only these solvers could solve the instances with 14 pigeons. The two methods could consistently improve the performance for instances with more than 11 pigeons. On the other hand, the effect of "m2" is moderate on MiniSAT. Even though it could shorten the processing time for all the instances with 11 pigeons, this is not the case for 12 pigeons. One of the possible reasons is that MiniSAT conducts less restarts than Glucose. While MiniSAT version 2.2 implements Luby restart [20] based on a static policy, Glucose version 3 executes a dynamic restart policy [21] which invokes restarting more frequently in practice. Thus, the method "m2" can be called more in Glucose than MiniSAT, which enhances the performance of Glucose. In contrast, the method "m1" also worked on MiniSAT. These results reveal that we can boost the performance of a SAT solver that implements methods to leverage extended variables aggressively for "fully" extended instances.

Table 1. Results of vanilla PHP instances.

Instance	n	#V	#C	#EV	#EC	M	M+m1	M+m2	G	G+m1	G+m2
no shuffle	11	110	561	0	0	100.39	100.41	**95.85**	122.81	121.48	122.72
shuffle1	11	110	561	0	0	105.26	104.96	**100.27**	119.39	117.89	119.26
shuffle2	11	110	561	0	0	46.23	46.24	**43.96**	52.39	51.97	52.42
shuffle3	11	110	561	0	0	95.92	95.78	**91.43**	104.67	103.37	103.97
no shuffle	12	132	738	0	0	935.58	934.61	882.17	519.32	**515.74**	532.06
shuffle1	12	132	738	0	0	1704.47	1711.09	1610.6	**558.32**	562.01	563.29
shuffle2	12	132	738	0	0	1744.06	1740.08	**1645.67**	3220.12	3215.18	3239.16
shuffle3	12	132	738	0	0	1709.55	1717.77	1616.86	**219.16**	219.7	221.88
no shuffle	13	156	949	0	0	10,000	10,000	10,000	**7684.53**	7694.96	7794.97
shuffle1	13	156	949	0	0	10,000	10,000	10,000	**4922.59**	4973.9	4947.7
shuffle2	13	156	949	0	0	10,000	10,000	10,000	5931.81	5918.43	**5914.93**
shuffle3	13	156	949	0	0	10,000	10,000	10,000	**5022.76**	5028.17	5054.09
no shuffle	14	182	1197	0	0	10,000	10,000	10,000	10,000	10,000	10,000
shuffle1	14	182	1197	0	0	10,000	10,000	10,000	10,000	10,000	10,000
shuffle2	14	182	1197	0	0	10,000	10,000	10,000	10,000	10,000	10,000
shuffle3	14	182	1197	0	0	10,000	10,000	10,000	10,000	10,000	10,000
no shuffle	15	210	1485	0	0	10,000	10,000	10,000	10,000	10,000	10,000
shuffle1	15	210	1485	0	0	10,000	10,000	10,000	10,000	10,000	10,000
shuffle2	15	210	1485	0	0	10,000	10,000	10,000	10,000	10,000	10,000
shuffle3	15	210	1485	0	0	10,000	10,000	10,000	10,000	10,000	10,000

Table 2. Results of PHP instances with full extension.

Instance	n	#V	#C	#EV	#EC	M	M+m1	M+m2	G	G+m1	G+m2
no shuffle	11	200	921	90	360	131.55	**4.17**	24.76	94.38	14.76	48.75
shuffle1	11	200	921	90	360	40.88	**3.12**	14.1	51.37	9.29	102.69
shuffle2	11	200	921	90	360	58.36	**2.76**	9.49	113.04	14.58	170.89
shuffle3	11	200	921	90	360	60.16	**2.92**	6.27	48.06	9.58	82.89
no shuffle	12	242	1178	110	440	1607.91	126.33	1806.44	1125.69	**91.54**	110.03
shuffle1	12	242	1178	110	440	2302.25	**55.9**	1249.96	748.22	61.34	86.71
shuffle2	12	242	1178	110	440	2411.08	138.41	1890.52	660.61	**57.07**	135.91
shuffle3	12	242	1178	110	440	2326.94	324.51	771.5	893.94	**70.4**	559.47
no shuffle	13	288	1477	132	528	10,000	2427.4	10,000	8565.22	**434.97**	478.05
shuffle1	13	288	1477	132	528	10,000	1045.38	10,000	3447.85	503.82	**364.8**
shuffle2	13	288	1477	132	528	10,000	2516.14	10,000	10,000	**282.12**	1679.05
shuffle3	13	288	1477	132	528	10,000	2702.69	10,000	3213.34	394.55	**250.65**
no shuffle	14	338	1821	156	624	10,000	10,000	10,000	10,000	6575.03	**4045.79**
shuffle1	14	338	1821	156	624	10,000	10,000	10,000	10,000	**2537.06**	3255.24
shuffle2	14	338	1821	156	624	10,000	10,000	10,000	10,000	**3006.74**	3374.63
shuffle3	14	338	1821	156	624	10,000	10,000	10,000	10,000	**2625.2**	3688.71
no shuffle	15	392	2213	182	728	10,000	10,000	10,000	10,000	10,000	10,000
shuffle1	15	392	2213	182	728	10,000	10,000	10,000	10,000	10,000	10,000
shuffle2	15	392	2213	182	728	10,000	10,000	10,000	10,000	10,000	10,000
shuffle3	15	392	2213	182	728	10,000	10,000	10,000	10,000	10,000	10,000

Table 3 exhibits the results of the PHP instances with partial extension. For this kind of instance, the two methods had good effects on MiniSAT but little impact on Glucose. MiniSAT with the methods stably outperformed the original MiniSAT for instances with 11 and 12 pigeons. On the other hand, Glucose with "m2" methods could not solve two instances with 13 pigeons that the original Glucose could solve. As mentioned in Section 4.1, partial extension can be useful for pruning the search space. However, in comparison with the results of full extension, partially extending variables cannot always help the solvers.

In these experiments, we can use already extended instances under the ideal extension. However, we have no idea how to add extended variables for various types of problems in practice. The results obtained from our experiments indicate that if we only derive partial extension during (or before) the search, the performance gain cannot be so significant. We

can have full benefits of extended resolution if we can fully (completely) conduct extension, which is not easy in practice.

Table 3. Results of PHP instances with partial extension.

Instance	n	#V	#C	#EV	#EC	M	M+m1	M+m2	G	G+m1	G+m2
no shuffle	11	119	597	9	36	101.48	51.14	**50.4**	105.33	65.76	78.55
shuffle1	11	119	597	9	36	45.8	101.2	**40.33**	79.43	122.02	44.11
shuffle2	11	119	597	9	36	98.69	97.87	**40.6**	170.46	105.53	91.8
shuffle3	11	119	597	9	36	44.82	100.02	**41.85**	56.57	47.55	108.75
no shuffle	12	142	778	10	40	2789.98	1774.52	693.75	1014.79	762.31	**565.39**
shuffle1	12	142	778	10	40	3293.2	1765.4	702.2	615.44	**592.57**	1253.2
shuffle2	12	142	778	10	40	2805.84	1815.72	703.13	622.15	922.09	**611.21**
shuffle3	12	142	778	10	40	1728.68	1837.56	705.31	674.8	**592.45**	759.31
no shuffle	13	167	993	11	44	10,000	10,000	10,000	10,000	**7442.11**	10,000
shuffle1	13	167	993	11	44	10,000	10,000	10,000	10,000	**6585.62**	10,000
shuffle2	13	167	993	11	44	10,000	10,000	10,000	7435.61	**5155.5**	10,000
shuffle3	13	167	993	11	44	10,000	10,000	10,000	**6573.76**	10,000	10,000
no shuffle	14	194	1245	12	48	10,000	10,000	10,000	10,000	10,000	10,000
shuffle1	14	194	1245	12	48	10,000	10,000	10,000	10,000	10,000	10,000
shuffle2	14	194	1245	12	48	10,000	10,000	10,000	10,000	10,000	10,000
shuffle3	14	194	1245	12	48	10,000	10,000	10,000	10,000	10,000	10,000
no shuffle	15	223	1537	13	52	10,000	10,000	10,000	10,000	10,000	10,000
shuffle1	15	223	1537	13	52	10,000	10,000	10,000	10,000	10,000	10,000
shuffle2	15	223	1537	13	52	10,000	10,000	10,000	10,000	10,000	10,000
shuffle3	15	223	1537	13	52	10,000	10,000	10,000	10,000	10,000	10,000

5.2. DRUP Analysis

Tables 4 and 5 exhibit the results of DRUP analysis of PHP instances with full extension for MiniSAT and Glucose with/without our methods. The second to fourth columns (n, #V, and #C) indicate the number of pigeons, variables (including extended variables), and clauses (including extended clauses), respectively. Note that solvers with a DRUP output run slower due to the overhead of recording DRUP, and instances with more than 13 pigeons are omitted since these instances could not be solved within the time limit (the "no-data" stands for the time limit exceeded one). As mentioned in Section 4.3, we count the frequencies (the number of appearances) of each clause in the unsatisfiability proof by analyzing DRUP proof of the instance. Then, we sort the frequencies in descending order and see the top-k ($k = 10$ and 100) of frequently used clauses for each instance. The values in columns top-k in the tables stand for the ratio of the number of extended clauses (clauses including at least one extended variable) in the top-k clauses. The column "ECR" indicates the ratio of the number of extended clauses to the number of whole clauses. Hence, if all the clauses uniformly contribute on the proof, the top-k ratio can be close to this rate. We can see that both solvers with the methods utilized lots of clauses including extended variables for almost all instances, while the original solvers hardly did. The column "p-value" stands for the result of the Wilcoxon rank-sum test [42], a nonparametric test of the null hypothesis that two sets of samples come from the same population. The first set is the clauses including (at least one) extended variables, and the second one is the clauses without extended variables. Note that in this test we observe all the clauses (not limited to top-k) used in the proof. In general, p-value less than 0.01 is considered to be statistically significant, which means that the null hypothesis should be rejected. In our experiments, we conduct the one-sided test where the alternative hypothesis is that clauses including extended variables are more frequently used in the proof. High correlation can be seen between the top-k ratio and p-value: higher top-k ratio gains lower p-value. From these results, we determine that our methods can firmly focus on the extended variables and shorten the processing time. Tables 6 and 7 exhibit the results of partial extension. Due to the few extended clauses (the ratio of extended clauses is under 0.06), these clauses appeared infrequently in the DRUP proof. Moreover, corresponding p-values are not low,

which does not indicate that clauses including extended variables contribute to the proof. While MiniSAT with the "m2" method utilized them for instances with 11 pigeons, the other solvers could hardly exploit the extended variables. Even always prioritizing the extended variables (by using the "m1" method), partial information of the ideal extension has little impact on the overall performance, at least by our proposed methods. The complete acquisition of extension can be necessary to gain practical speedup of the solvers.

Table 4. DRUP analysis results of PHP instances with full extension for MiniSAT.

					M			M+m1			M+m2		
Instance	**n**	**#V**	**#C**	**ECR**	**top-10**	**top-100**	***p*-Value**	**top-10**	**top-100**	***p*-Value**	**top-10**	**top-100**	***p*-Value**
no shuffle	11	200	921	0.39	0.00	0.00	1.00×10^{0}	0.80	0.77	1.03×10^{-85}	0.50	0.51	1.09×10^{-34}
shuffle1	11	200	921	0.39	0.00	0.00	1.00×10^{0}	0.10	0.32	6.73×10^{-1}	0.70	0.81	7.01×10^{-34}
shuffle2	11	200	921	0.39	0.00	0.00	1.00×10^{0}	0.50	0.74	6.44×10^{-46}	0.70	0.76	1.42×10^{-21}
shuffle3	11	200	921	0.39	0.00	0.00	1.00×10^{0}	0.50	0.77	6.73×10^{-53}	0.60	0.61	6.31×10^{-2}
no shuffle	12	242	1178	0.37	0.00	0.00	1.00×10^{0}	0.80	0.59	7.40×10^{-67}	0.70	0.80	5.12×10^{-57}
shuffle1	12	242	1178	0.37	0.00	0.00	1.00×10^{0}	0.60	0.82	2.28×10^{-24}	0.50	0.55	1.79×10^{-2}
shuffle2	12	242	1178	0.37	0.00	0.00	1.00×10^{0}	0.30	0.31	3.07×10^{-1}	0.00	0.00	1.00×10^{0}
shuffle3	12	242	1178	0.37	0.00	0.00	1.00×10^{0}	0.00	0.68	1.67×10^{-18}	0.60	0.82	1.03×10^{-32}
no shuffle	13	288	1477	0.36	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data
shuffle1	13	288	1477	0.36	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data
shuffle2	13	288	1477	0.36	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data
shuffle3	13	288	1477	0.36	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data

Table 5. DRUP analysis results of PHP instances with full extension for Glucose.

					G			G+m1			G+m2		
Instance	**n**	**#V**	**#C**	**ECR**	**top-10**	**top-100**	***p*-Value**	**top-10**	**top-100**	***p*-Value**	**top-10**	**top-100**	***p*-Value**
no shuffle	11	200	921	0.39	0.00	0.36	6.20×10^{-23}	0.70	0.79	9.77×10^{-64}	0.10	0.64	1.07×10^{-38}
shuffle1	11	200	921	0.39	0.00	0.00	2.11×10^{-1}	0.70	0.80	8.08×10^{-58}	0.00	0.22	1.00×10^{0}
shuffle2	11	200	921	0.39	0.00	0.00	1.00×10^{0}	0.60	0.78	7.50×10^{-45}	0.10	0.37	5.72×10^{-1}
shuffle3	11	200	921	0.39	0.00	0.00	7.89×10^{-1}	0.70	0.76	4.72×10^{-63}	0.00	0.08	1.00×10^{0}
no shuffle	12	242	1178	0.37	0.00	0.00	2.93×10^{-7}	0.60	0.85	3.27×10^{-168}	0.20	0.65	4.96×10^{-55}
shuffle1	12	242	1178	0.37	0.00	0.00	1.00×10^{0}	0.70	0.80	1.55×10^{-45}	0.00	0.54	4.38×10^{-39}
shuffle2	12	242	1178	0.37	0.00	0.00	1.00×10^{0}	0.60	0.80	3.98×10^{-59}	0.50	0.71	1.51×10^{-67}
shuffle3	12	242	1178	0.37	0.00	0.00	9.99×10^{-1}	0.60	0.85	8.20×10^{-68}	0.00	0.12	4.10×10^{-17}
no shuffle	13	288	1477	0.36	no-data	no-data	no-data	0.90	0.90	4.53×10^{-215}	0.20	0.48	1.09×10^{-69}
shuffle1	13	288	1477	0.36	0.00	0.00	9.97×10^{-1}	0.80	0.82	1.77×10^{-46}	0.70	0.86	1.30×10^{-72}
shuffle2	13	288	1477	0.36	no-data	no-data	no-data	0.60	0.85	6.33×10^{-83}	0.30	0.51	2.85×10^{-19}
shuffle3	13	288	1477	0.36	0.00	0.00	9.04×10^{-1}	0.60	0.82	2.34×10^{-72}	0.80	0.83	1.88×10^{-64}

Table 6. DRUP analysis results of PHP instances with partial extension for MiniSAT.

					M			M+m1			M+m2		
Instance	**n**	**#V**	**#C**	**ECR**	**top-10**	**top-100**	***p*-Value**	**top-10**	**top-100**	***p*-Value**	**top-10**	**top-100**	***p*-Value**
no shuffle	11	119	597	0.06	0.00	0.00	1.00×10^{0}	0.10	0.02	1.83×10^{-2}	0.00	0.00	6.47×10^{-2}
shuffle1	11	119	597	0.06	0.00	0.00	1.00×10^{0}	0.00	0.00	7.09×10^{-1}	0.30	0.06	1.00×10^{0}
shuffle2	11	119	597	0.06	0.00	0.00	1.00×10^{0}	0.10	0.01	7.23×10^{-1}	0.20	0.04	9.85×10^{-1}
shuffle3	11	119	597	0.06	0.00	0.00	1.00×10^{0}	0.20	0.04	7.86×10^{-1}	0.10	0.03	7.56×10^{-1}
no shuffle	12	142	778	0.05	0.00	0.00	1.00×10^{0}	0.00	0.00	1.00×10^{0}	0.00	0.00	1.58×10^{-9}
shuffle1	12	142	778	0.05	0.00	0.00	1.00×10^{0}	0.00	0.00	1.00×10^{0}	0.00	0.02	1.00×10^{0}
shuffle2	12	142	778	0.05	0.00	0.00	1.00×10^{0}	0.00	0.00	1.00×10^{0}	0.00	0.02	9.25×10^{-1}
shuffle3	12	142	778	0.05	0.00	0.00	1.00×10^{0}	0.00	0.00	1.00×10^{0}	0.00	0.04	6.00×10^{-2}
no shuffle	13	167	993	0.04	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data
shuffle1	13	167	993	0.04	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data
shuffle2	13	167	993	0.04	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data
shuffle3	13	167	993	0.04	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data

Table 7. DRUP analysis results of PHP instances with partial extension for Glucose.

					G			G+m1			G+m2		
Instance	**n**	**#V**	**#C**	**ECR**	**top-10**	**top-100**	***p*-Value**	**top-10**	**top-100**	***p*-Value**	**top-10**	**top-100**	***p*-Value**
no shuffle	11	119	597	0.06	0.00	0.00	1.00×10^{0}	0.10	0.01	1.29×10^{-1}	0.10	0.27	2.74×10^{-11}
shuffle1	11	119	597	0.06	0.00	0.00	1.00×10^{0}	0.00	0.02	9.26×10^{-2}	0.00	0.03	8.60×10^{-1}
shuffle2	11	119	597	0.06	0.00	0.00	9.98×10^{-1}	0.10	0.02	7.78×10^{-1}	0.00	0.00	1.00×10^{0}
shuffle3	11	119	597	0.06	0.00	0.00	1.00×10^{0}	0.00	0.00	6.82×10^{-1}	0.00	0.00	1.00×10^{0}
no shuffle	12	142	778	0.05	0.00	0.00	3.73×10^{-1}	0.10	0.01	7.14×10^{-3}	0.00	0.00	9.98×10^{-1}
shuffle1	12	142	778	0.05	0.00	0.00	1.00×10^{0}	0.00	0.00	9.52×10^{-1}	0.00	0.00	9.98×10^{-1}
shuffle2	12	142	778	0.05	0.00	0.00	1.00×10^{0}	0.10	0.02	4.89×10^{-1}	0.00	0.02	4.69×10^{-1}
shuffle3	12	142	778	0.05	0.00	0.00	1.00×10^{0}	0.00	0.00	3.23×10^{-1}	0.10	0.04	4.29×10^{-2}
no shuffle	13	167	993	0.04	no-data	no-data	no-data	0.10	0.02	1.00×10^{0}	no-data	no-data	no-data
shuffle1	13	167	993	0.04	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data	no-data
shuffle2	13	167	993	0.04	0.00	0.00	1.00×10^{0}	0.10	0.01	1.38×10^{-1}	no-data	no-data	no-data
shuffle3	13	167	993	0.04	0.00	0.00	9.94×10^{-1}	no-data	no-data	no-data	no-data	no-data	no-data

6. Conclusions

In theory, extended resolution enables polynomial solving of the pigeonhole principle. However, few works have clarified the effectiveness of extended variables on today's CDCL (i.e., general resolution-based algorithm) SAT solvers. We investigated the "practical effect" of the extended resolution on the pigeonhole principle by prioritizing the extended variables in the solver's search in two ways. Combining with these methods, the PHP instances with the extended variables were solved within a shorter processing time by MiniSAT 2.2 and Glucose 3. While fully extended instances had clearer performance gain, partially extended ones had moderate effects. Even though it is not clear our methods could fully utilize the extended variables, our results indicate that we cannot attain good performance gain on the partially extended instances. Note that the method of extension is under the ideal condition, and there cannot be definite guidelines on how to add the extended variables in practice. Although it is difficult to add appropriate auxiliary variables and utilize them, our results exhibit the possibility of performance enhancement by extension. We believe that the facts in this paper create a stir to the practical usefulness of the extended resolution for instances other than the PHP.

Funding: This work was supported by JSPS KAKENHI Grant Number JP17K12742 and JP21K12023.

Institutional Review Board Statement: Not applicale.

Data Availability Statement: The datasets generated during the current study are not publicly available but are available from the corresponding author on reasonable request.

Acknowledgments: We appreciate all the researchers giving helpful comments on this paper.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Code of Our Proposed Methods

In this section, we show the actual code of our proposed methods, "m1" and "m2". Note that the code of Glucose is based on MiniSAT, thus the codes can work in both solvers.

We show the source code of the "m1" method in Listings A1 and A2. Listing A1 is embedded in Solver.h file of MiniSAT/Glucose, and Listing A2 is embedded in Solver.cc file (head and tail parts are the same as the original code and omitted). By extremely increasing the VSIDS scores of extended variables, the solver always selects them as decision variables prior to non-extended variables.

Listing A1. Code of m1 method implemented in Solver.h.

```
int num_org_vars;         // calculated from the number of pigeons
double evar_inc = 1e100; // incremental score for extended variables
void bumpExtendedVars(
  const int num_org_vars, const int curr_restarts
){
  if(curr_restarts != 0)return;
  const int nv = nVars();
  for(int i = num_org_vars; i < nv; i++){
    varBumpActivity(i, evar_inc);
  }
}

inline void Solver::varBumpActivity(Var v, double inc){
  if(v < num_org_vars)
    activity[v] += inc;
  else
    activity[v] += inc * evar_inc;
  if((v < num_org_vars && activity[v] > 1e100) ||
    (v >= num_org_vars && activity[v] > 1e200)){
    // Rescale:
    for(int i = 0; i < nVars(); i++)
      activity[i] *= 1e-100;
    var_inc *= 1e-100;
  }
  // Update order_heap with respect to new activity:
  if(order_heap.inHeap(v))
    order_heap.decrease(v);
}
```

Listing A2. Code of m1 method implemented in Solver.cc.

```
lbool Solver::solve_(){
  ...
  int curr_restarts = 0;
  bumpExtendedVars(num_org_vars, curr_restarts);
  while (status == l_Undef){
    double rest_base = luby_restart ? luby(restart_inc, curr_restarts) :
                                      pow(restart_inc, curr_restarts);
    status = search(rest_base * restart_first);
  ...
}
```

We show the source code of the "m2" method in Listings A3 and A4. Like the "m1" method, Listing A1 is embedded in Solver.h file of MiniSAT/Glucose, and Listing A2 is embedded in Solver.cc file (head and tail parts are also omitted). Similar to "m1", VSIDS scores of extended variables are periodically (i.e., at every restart) increased.

Listing A3. Code of m2 method implemented in Solver.h.

```
void bumpExtendedVars(const int num_org_vars){
  for(int i = num_org_vars; i < nVars(); i++){
    varBumpActivity(i, var_inc * 100);
  }
}
```

Listing A4. Code of m2 method implemented in Solver.cc.

```
lbool Solver::solve_(){
  ...
  int curr_restarts = 0;
  int num_org_vars;        // calculated from the number of pigeons
  while (status == l_Undef){
    bumpExtendedVars(num_org_vars, curr_restarts);
    double rest_base = luby_restart ? luby(restart_inc, curr_restarts) :
                                      pow(restart_inc, curr_restarts);
    status = search(rest_base * restart_first);
  ...
}
```

References

1. Cook, S.A. The Complexity of Theorem-Proving Procedures. In Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, Shaker Heights, OH, USA, 3–5 May 1971; pp. 151–158. [CrossRef]
2. Stephan, P.; Brayton, R.K.; Sangiovanni-Vincentelli, A.L. Combinational test generation using satisfiability. *Trans. Comput. Aided Des. Integr. Circuits Syst.* **2006**, *15*, 1167–1176. [CrossRef]
3. Narodytska, N.; Kasiviswanathan, S.P.; Ryzhyk, L.; Sagiv, M.; Walsh, T. Verifying Properties of Binarized Deep Neural Networks. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, LA, USA, 2–7 February 2018; pp. 6615–6624. [CrossRef]
4. Heule, M.J.H.; Kullmann, O.; Marek, V.W. Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer. In Proceedings of the Theory and Applications of Satisfiability Testing-SAT 2016-19th International Conference, Bordeaux, France, 5–8 July 2016; pp. 228–245. [CrossRef]
5. Heule, M.J.H. Schur Number Five. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, LA, USA, 2–7 February 2018; pp. 6598–6606. [CrossRef]
6. Bright, C.; Kotsireas, I.S.; Heinle, A.; Ganesh, V. Complex Golay pairs up to length 28: A search via computer algebra and programmatic SAT. *J. Symb. Comput.* **2021**, *102*, 153–172. [CrossRef]
7. Davis, M.; Logemann, G.; Loveland, D. A machine program for theorem-proving. *Commun. ACM* **1962**, *5*, 394–397. [CrossRef]
8. Silva, J.P.M.; Sakallah, K.A. GRASP—A new search algorithm for satisfiability. In Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, 10–14 November 1996; pp. 220–227. [CrossRef]
9. Tseitin, G.S. On the complexity of derivation in propositional calculus. In *Automation of 468 Reasoning: 2: Classical Papers on Computational Logic 1967–1970*; Springer: Berlin/Heidelberg, Germany, 1983; pp. 466–483. [CrossRef]
10. Cook, S.A. A short proof of the pigeon hole principle using extended resolution. *ACM Sigact News* **1976**, *8*, 28–32. [CrossRef]
11. Huang, J. Extended clause learning. *Artif. Intell.* **2010**, *174*, 1277–1284. [CrossRef]
12. Audemard, G.; Katsirelos, G.; Simon, L. A Restriction of Extended Resolution for Clause Learning SAT Solvers. In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, GA, USA, 11–15 July 2010. [CrossRef]
13. Heule, M.H., Jr.; Hunt, W.A.; Wetzler, N. Trimming while checking clausal proofs. In Proceedings of the Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013; pp. 181–188. [CrossRef]
14. Eén, N.; Sörensson, N. An Extensible SAT-solver. In Proceedings of the Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, 5–8 May 2003; pp. 502–518. [CrossRef]
15. Audemard, G.; Simon, L. Predicting Learnt Clauses Quality in Modern SAT Solvers. In Proceedings of the IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, CA, USA, 11–17 July 2009; pp. 399–404.
16. Ignatiev, A.; Morgado, A.; Marques-Silva, J. On Tackling the Limits of Resolution in SAT Solving. In Proceedings of the Theory and Applications of Satisfiability Testing-SAT 2017-20th International Conference, Melbourne, Australia, 28 August–1 September 2017; pp. 164–183. [CrossRef]
17. Moskewicz, M.W.; Madigan, C.F.; Zhao, Y.; Zhang, L.; Malik, S. Chaff: Engineering an Efficient SAT Solver. In Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, 18–22 June 2001; pp. 530–535. [CrossRef]
18. Bayardo, R.J., Jr.; Schrag, R. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, Providence, RI, USA, 27–31 July 1997; pp. 203–208. [CrossRef]
19. Gomes, C.P.; Selman, B.; Kautz, H.A. Boosting Combinatorial Search Through Randomization. In Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, Madison, WI, USA, 26–30 July 1998; pp. 431–437. [CrossRef]
20. Huang, J. The Effect of Restarts on the Efficiency of Clause Learning. In Proceedings of the IJCAI 2007, 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, 6–12 January 2007; pp. 2318–2323. [CrossRef]

21. Audemard, G.; Simon, L. Refining Restarts Strategies for SAT and UNSAT. In Proceedings of the Principles and Practice of Constraint Programming-18th International Conference, CP 2012, Québec City, QC, Canada, 8–12 October 2012; Volume 7514, pp. 118–126. [CrossRef]
22. Liang, J.H.; Ganesh, V.; Poupart, P.; Czarnecki, K. Learning Rate Based Branching Heuristic for SAT Solvers. In Proceedings of the Theory and Applications of Satisfiability Testing-SAT 2016-19th International Conference, Bordeaux, France, 5–8 July 2016; pp. 123–140. [CrossRef]
23. Cherif, M.S.; Habet, D.; Terrioux, C. Combining VSIDS and CHB Using Restarts in SAT. In Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France, 25–29 October 2021; pp. 20:1–20:19. [CrossRef]
24. Cai, S.; Zhang, X. Deep Cooperation of CDCL and Local Search for SAT. In Proceedings of the Theory and Applications of Satisfiability Testing-SAT 2021-24th International Conference, Barcelona, Spain, 5–9 July 2021; pp. 64–81. [CrossRef]
25. Bacchus, F.; Winter, J. Effective Preprocessing with Hyper-Resolution and Equality Reduction. In Proceedings of the Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, 5–8 May 2003; pp. 341–355. [CrossRef]
26. Eén, N.; Biere, A. Effective Preprocessing in SAT Through Variable and Clause Elimination. In Proceedings of the Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, 19–23 June 2005; pp. 61–75. [CrossRef]
27. Heule, M.; Järvisalo, M.; Biere, A. Efficient CNF Simplification Based on Binary Implication Graphs. In Proceedings of the Theory and Applications of Satisfiability Testing-SAT 2011-14th International Conference, SAT 2011, Ann Arbor, MI, USA, 19–22 June 2011; pp. 201–215. [CrossRef]
28. Järvisalo, M.; Heule, M.; Biere, A. Inprocessing Rules. In Proceedings of the Automated Reasoning-6th International Joint Conference, IJCAR 2012, Manchester, UK, 26–29 June 2012; pp. 355–370. [CrossRef]
29. Cook, S.A.; Reckhow, R.A. The relative efficiency of propositional proof systems. *J. Symb. Log.* **1979**, *44*, 36–50. [CrossRef]
30. Sinz, C.; Biere, A. Extended Resolution Proofs for Conjoining BDDs. In Proceedings of the Computer Science-Theory and Applications, First International Computer Science Symposium in Russia, CSR 2006, St. Petersburg, Russia, 8–12 June 2006; pp. 600–611. [CrossRef]
31. Kullmann, O. On a generalization of extended resolution. *Discret. Appl. Math.* **1999**, *96*, 149–176. [CrossRef]
32. Järvisalo, M.; Biere, A.; Heule, M. Blocked Clause Elimination. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, 20–28 March 2010; pp. 129–144. [CrossRef]
33. Wetzler, N.; Heule, M.J.H.; Hunt, W.A., Jr. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In Proceedings of the Theory and Applications of Satisfiability Testing-SAT 2014-17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, 14–17 July 2014; pp. 422–429. [CrossRef]
34. Kiesl, B.; Rebola-Pardo, A.; Heule, M.J.H. Extended Resolution Simulates DRAT. In Proceedings of the Automated Reasoning-9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, 14–17 July 2018; pp. 516–531. [CrossRef]
35. Kiesl, B.; Rebola-Pardo, A.; Heule, M.J.; Biere, A. Simulating strong practical proof systems with extended resolution. *J. Autom. Reason.* **2020**, *64*, 1247–1267. [CrossRef]
36. Huang, J. A Case for Simple SAT Solvers. In Proceedings of the Principles and Practice of Constraint Programming-CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, 23–27 September 2007; pp. 839–846. [CrossRef]
37. Manthey, N. Extended resolution in modern SAT solving. In Proceedings of the Joint Automated Reasoning Workshop and Deduktionstreffen: As part of the Vienna Summer of Logic—IJCAR, Vienna, Austria, 23–24 July 2014; pp. 26–27.
38. Jabbour, S.; Lonlac, J.; Sais, L. Extending Resolution by Dynamic Substitution of Boolean Functions. In Proceedings of the IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, 7–9 November 2012; pp. 1029–1034. [CrossRef]
39. Manthey, N.; Heule, M.; Biere, A. Automated Reencoding of Boolean Formulas. In Proceedings of the Hardware and Software: Verification and Testing-8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, 6–8 November 2012; pp. 102–117. [CrossRef]
40. Manthey, N. Coprocessor 2.0—A Flexible CNF Simplifier-(Tool Presentation). In Proceedings of the Theory and Applications of Satisfiability Testing-SAT 2012-15th International Conference, Trento, Italy, 17–20 June 2012; pp. 436–441. [CrossRef]
41. Simon, L. Post Mortem Analysis of SAT Solver Proofs. In Proceedings of the POS-14. Fifth Pragmatics of SAT Workshop, a Workshop of the SAT 2014 Conference, Part of FLoC 2014 during the Vienna Summer of Logic, Vienna, Austria, 13 July 2014; pp. 26–40. [CrossRef]
42. Mann, H.B.; Whitney, D.R. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Stat.* **1947**, *18*, 50–60. [CrossRef]