XOR Local Search for Boolean Brent Equations

Wojciech Nawrocki[1], Zhenjun Liu[1], Andreas Fröhlich[2], Marijn J.H. Heule[1], and Armin Biere[2]

[1] Carnegie Mellon University, Pittsburgh, United States
{wnawrock, zhenjunl, mheule}@andrew.cmu.edu
[2] Johannes Kepler University, Linz, Austria
{andreas.froehlich, biere}@jku.at

Abstract. Combining clausal and XOR reasoning has been studied for almost two decades, in particular in the context of CDCL and look-ahead, but not in classical local search. To stimulate research in this direction, we propose to standardize a hybrid format, called XNF, which allows both clauses and XORs. We implemented a tool to extract XOR constraints from a CNF, simplify them, and produce an XNF formula. The usefulness of XNF formulas is demonstrated by focusing on the impact of combined clausal and XOR reasoning on local search. Native support for XOR facilitates satisfying any falsified long XOR using a single flip, similarly to satisfying a falsified clause. When combined with XOR-based heuristics, local search performance is significantly improved on matrix multiplication challenges which are hard for CDCL.

1 Introduction

Two of the most successful approaches to SAT solving are Conflict-Driven Clause Learning (CDCL) and Stochastic Local Search (SLS). Modern CDCL solvers are very sophisticated and able to efficiently solve a broad range of problems. In contrast, the idea of SLS is simple yet works well on certain formulas. Also look-ahead solvers have been quite successful in the past, but suffer from having few applications that are not already successfully covered by CDCL.

These solving paradigms usually operate on conjunctive normal form (CNF) and thus expect their input to be a set of clauses. While in principle all problems can be translated into pure CNF, additionally allowing the use of XOR constraints can provide a more natural representation, which in turn can possibly lead to more efficient solving approaches. Examples include problems from cryptography, with corresponding formulas often originally denoted in algebraic normal form, but also all formulas that simply contain parity constraints [4,5,18,31]. Similarly, XOR constraints are important for approximate model counting [13]. As a result, the combination of clausal and XOR reasoning has been considered an interesting topic and has been studied [37] as well as applied in several algorithms—usually in the context of CDCL and look-ahead solvers [14,20], with

CryptoMiniSAT probably being the most prominent example [38]. Nevertheless, most state-of-the-art solvers still do not support XOR reasoning.

Research on XOR constraints in the context of SLS is sparse and, aside from loosely related work on gates [6,33], satisfiability modulo theories [17], and continuous local search [29,30], there have been few successful attempts on improving SLS solvers by incorporating support for non-CNF representations. In particular, up-to-date there is no SLS algorithm that combines clausal and XOR reasoning during the search process by supporting native XOR constraints.

Moreover, SLS solvers have different strengths than other types of solvers. For instance, they are considered to work well on random $k-$SAT and satisfiable, hard combinatorial problems. On some crafted combinatorial problems such as `VanDerWaeden_pd_3k` and `battleship`, the SLS solver `Swcca` outperforms the CDCL solver `Glucose` in both success rate and average time [12]. Another well-known example is the boolean Pythagorean Triples problem [23]: the satisfiable $[1, 7824]$ instance can be solved using DDFW local search [24] in one CPU minute, while complete methods can take years.

A recent problem of interest on which SLS performed particularly well is related to matrix multiplication [21] expressed as a SAT problem using Boolean Brent equations [22]. The corresponding matrix multiplication (challenge 1) benchmarks[3], `MM-Challenge-1`, turn out to be hard for CDCL solvers, but could be solved by the SLS solver `YalSAT`. This is particularly surprising since the benchmark formulas contain a large number of XOR constraints which, encoded into CNF, should (and actually do) hinder the performance of SLS solvers.

Why do we consider CNF-encoded XOR constraints to be problematic for SLS solvers? To avoid combinatorial explosion in the number of clauses, the Tseitin transformation has to be used, particularly for long XOR constraints. While shortening the formula, this encoding introduces a large number of auxiliary variables which, roughly speaking, obscure the XOR constraint from the solver's view and drastically affect the neighbourhood of assignments visited during the local search. We will give a detailed explanation in Section 2. This observation, together with the already good performance of SLS on `MM-Challenge-1`, provides even more reason to assume that it might be possible to further push the state-of-the-art by incorporating native XOR support into SLS.

Our contribution. The core observation on which we base our work is that XOR constraints fit quite naturally into SLS algorithms. Every time a literal is flipped, the truth values of all XOR constraints containing the literal get flipped as well, and the core solver loop can be adapted to do this. To support this, we extended our input format from CNF to XNF, allowing XORs to exist as another type of constraint alongside the usual disjunctive clauses. We also developed a tool `cnf2xnf`, which extracts XOR constraints from a formula given in CNF and saves the result in XNF format, as well as a related tool `extor` which reconstructs solutions for the original CNF from solutions for the XNF. Both algorithms are described in Section 3. Our main contribution is `xnfSAT`, an SLS solver based on the state-of-the-art `YalSAT` [8] solver—we outline its implementation

[3] https://github.com/marijnheule/matrix-challenges

in Section 4. We then present experimental results in Section 5 and show that `xnfSAT` achieves significant performance improvements on all benchmarks within `MM-Challenge-1`, thus confirming the usefulness of our combined representation and pushing the state-of-the-art on these challenging instances.

2 XOR Constraints

An SLS solver starts with a complete assignment of truth values to variables. While the formula is not satisfied, it loops flipping literals chosen according to some probability distribution. The choice of this distribution forms the heart of an SLS solver [2].

To see why CNF-encoded XOR constraints can negatively impact the performance of SLS solvers, let us first summarize briefly a simplified version of the SLS algorithm as used in `YalSAT` [8].

Algorithm 1 outlines the high-level structure of `YalSAT`, omitting certain details such as restarts and corresponding strategy changes. The basic loop originates from `WalkSAT` [34]. The solver first builds up internal data structures, preprocesses the formula via unit propagation, and sets an initial truth-value assignment. It then loops until a solution is found. On each iteration, it picks a falsified clause and flips the truth value of a literal in it. Details of this process will be outlined in Section 4—for now, a bird's eye view suffices.

Algorithm 1 Outline of a typical `WalkSAT`-based solver

```
1: for clause in input file do
2:     parse and store clause to data structure
3: end for
4: preprocess formula
5: α ← complete initial assignment of truth values
6: while there exists a clause falsified by α do
7:     C ← pickUnsatClause()
8:     x ← pickVar(C)
9:     α ← α with x flipped
10:    update solver state
11: end while
```

CNF Encodings

Let us now look at how this algorithm interacts with the CNF encoding of XOR constraints. The direct encoding of an XOR constraint on k variables uses 2^{k-1} clauses of length k, where each clause consists of variables $x_1, \ldots, x_k$ with an even number of them negated:

$$\texttt{XOR\_d}(x_1, x_2, \ldots, x_k) = \bigwedge_{\text{even } \#\neg} (\pm x_1 \vee \pm x_2 \vee \cdots \vee \pm x_k)$$

To avoid an exponential growth in the number of clauses, it is common to use the Tseitin transformation [40] instead, which recursively translates arbitrary formulas into CNF by introducing fresh auxiliary variables for its sub-formulas:

$$f(g(x_1, \ldots, x_k)) = f(y) \wedge (y \leftrightarrow g(x_1, \ldots, x_k))$$

The resulting formula is equisatisfiable to the original one. Due to its recursive nature and the associative property of certain binary operations, the final CNF representation can differ in the number of variables and clauses depending on the structure of the original formula.

We consider two different parameters that describe this structure in the case of pure XOR constraints and, thus, influence the final CNF representation: the *cutting number* and the *mode*.

The cutting number, roughly speaking, defines the size of the individual slices that are cut out of an XOR constraint and then encoded in a direct fashion [4]. The smaller the slices (with a minimum of size $n = 3$), the shorter and fewer the resulting clauses, since each slice will be encoded into 2^{n-1} clauses of length n. However, with larger n, fewer slices are required and, thus, fewer auxiliary variables need to be introduced.

It is not clear whether there is a universally optimal setting for the cutting number. Soos and Meel [37] argue that a cutting number of 4 experimentally turned out to be optimal for their use case in approximate model counting. In contrast, for some problems in cryptography, it is suggested that a cutting number of 6 would be the optimal setting for the respective applications [5,10]. In Section 5, we analyze results for `xnfSAT` on CNF benchmarks constructed using several cutting numbers.

The second parameter, which we call the mode of translation, influences the fashion in which the XOR constraint is recursively traversed. For the mode, we distinguish between *linear* and *pooled* encodings. While the linear mode might be considered the standard approach and has been used before [5], we are not aware of any previous work using the pooled mode—however, a similar approach for at-most-one constraints has been contributed to Knuth's book "The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability" [27] (p. 134, Ex. 12) by the fourth author.

From an implementational point of view, the difference between the two modes basically boils down to whether a stack or a queue is used when removing a chunk of variables from the XOR constraint and adding fresh auxiliary variables during translation. In particular, a stack will lead to a linear translation, whereas a queue will produce a pooled one. We can assume $k > n$, with k being the length of the XOR constraint and n denoting the cutting number. If this were not the case, we could simply use the direct encoding. Let `XOR_l_n` and `XOR_p_n` denote the linear and the pooled encoding, respectively, with cutting number n:

$$\texttt{XOR\_l\_n}(x_1, \ldots, x_k) = \texttt{XOR\_d}(x_1, \ldots, x_{n-1}, \overline{y}) \wedge \texttt{XOR\_l\_n}(y, x_n, \ldots, x_k)$$
$$\texttt{XOR\_p\_n}(x_1, \ldots, x_k) = \texttt{XOR\_d}(x_1, \ldots, x_{n-1}, \overline{y}) \wedge \texttt{XOR\_p\_n}(x_n, \ldots, x_k, y)$$

Note that XOR chunks (with length $n - 1$) are always sliced from the left. The new auxiliary variable in linear mode or pooled mode is then added to the left or

the right of the remaining XOR constraint, respectively. For the sliced chunks, the position of $\overline{y}$ does not matter since the direct encoding is not affected by variable order. To illustrate the practical difference of the two modes, let us take a closer look at a short example for an XOR constraint of length 6 (for simplicity, using a fixed cutting number of 3).

Example 1.

$$\begin{aligned}
&\texttt{XOR_l_3}(x_1, x_2, x_3, x_4, x_5, x_6)\\
&\quad = \texttt{XOR_d}(x_1, x_2, \overline{y_1}) \wedge \texttt{XOR_l_3}(y_1, x_3, x_4, x_5, x_6)\\
&\quad = \texttt{XOR_d}(x_1, x_2, \overline{y_1}) \wedge \texttt{XOR_d}(y_1, x_3, \overline{y_2}) \wedge \texttt{XOR_l_3}(y_2, x_4, x_5, x_6)\\
&\quad = \texttt{XOR_d}(x_1, x_2, \overline{y_1}) \wedge \texttt{XOR_d}(y_1, x_3, \overline{y_2}) \wedge \texttt{XOR_d}(y_2, x_4, \overline{y_3}) \wedge \texttt{XOR_l_3}(y_3, x_5, x_6)\\
&\quad = \texttt{XOR_d}(x_1, x_2, \overline{y_1}) \wedge \texttt{XOR_d}(y_1, x_3, \overline{y_2}) \wedge \texttt{XOR_d}(y_2, x_4, \overline{y_3}) \wedge \texttt{XOR_d}(y_3, x_5, x_6)
\end{aligned}$$

$$\begin{aligned}
&\texttt{XOR_p_3}(x_1, x_2, x_3, x_4, x_5, x_6)\\
&\quad = \texttt{XOR_d}(x_1, x_2, \overline{y_1}) \wedge \texttt{XOR_p_3}(x_3, x_4, x_5, x_6, y_1)\\
&\quad = \texttt{XOR_d}(x_1, x_2, \overline{y_1}) \wedge \texttt{XOR_d}(x_3, x_4, \overline{y_2}) \wedge \texttt{XOR_p_3}(x_5, x_6, y_1, y_2)\\
&\quad = \texttt{XOR_d}(x_1, x_2, \overline{y_1}) \wedge \texttt{XOR_d}(x_3, x_4, \overline{y_2}) \wedge \texttt{XOR_d}(x_5, x_6, \overline{y_3}) \wedge \texttt{XOR_p_3}(y_1, y_2, y_3)\\
&\quad = \texttt{XOR_d}(x_1, x_2, \overline{y_1}) \wedge \texttt{XOR_d}(x_3, x_4, \overline{y_2}) \wedge \texttt{XOR_d}(x_5, x_6, \overline{y_3}) \wedge \texttt{XOR_d}(y_1, y_2, y_3)
\end{aligned}$$

While the structure of the resulting CNF formula (for different modes) as well as the number of variables and clauses (for different cutting numbers) will vary, all combinations are effective in reducing the number of clauses at the expense of adding linearly more variables—a relatively small price to pay. Nevertheless, this translation can greatly hinder the performance of local search solvers.

To see this, consider $\texttt{XOR\_l\_n}(x_1, \ldots, x_k)$. For original variables $x_1, \ldots, x_k$, the encoding introduces auxiliary variables $y_1, \ldots, y_r$ (with $r \in \Theta(k)$) and $r - 1$ XOR constraints of length $n < k$. For simplicity, let us again assume $n = 3$:

$$\texttt{XOR\_l\_3}(x_1, .., x_k) = \texttt{XOR\_d}(x_1, x_2, \overline{y_1}) \wedge \texttt{XOR\_d}(y_1, x_3, \overline{y_2}) \wedge \cdots \wedge \texttt{XOR\_d}(y_r, x_{k-1}, x_k)$$

Observe that for each assignment of $x_1, \ldots, x_k$ satisfying $\texttt{XOR\_d}(x_1, \ldots, x_k)$, there exists a unique assignment of $y_1, \ldots, y_r$ that satisfies $\texttt{XOR\_l\_3}(x_1, \ldots, x_k)$. This is because, given an assignment of $x_1, \ldots, x_k$ satisfying $\texttt{XOR\_d}(x_1, \ldots, x_k)$, there is only one assignment of y_1 satisfying $\texttt{XOR\_d}(x_1, x_2, \overline{y_1})$, which subsequently forces the assignment of y_2 in order to satisfy $\texttt{XOR}(y_1, x_3, \overline{y_2})$, and so on. The same kind of argument holds for the general encodings $\texttt{XOR\_l\_n}$ and $\texttt{XOR\_p\_n}$.

We say that an assignment satisfies the CNF-encoded XOR constraint (by $\texttt{XOR\_l\_n}$ or $\texttt{XOR\_p\_n}$) *on a high level* if an odd number of $x_1, \ldots, x_k$ are set to true (i.e., if the original XOR constraint would be satisfied). However, even when the constraint is satisfied on a high level, it is possible that the auxiliary variables do not have the correct unique values. In this way, an XOR constraint can be satisfied on a high level but falsified in the (low-level) Tseitin CNF encoding.

For the SLS solver to move from a falsifying assignment of $x_1, \ldots, x_k$ to a satisfying assignment of $x_1, \ldots, x_k$, it additionally needs to flip the correct

auxiliary variables to match the corresponding assignment of $y_1, \ldots, y_r$. However, there is only one assignment of $y_1, \ldots, y_r$ satisfying the Tseitin-encoded CNF representation of $(x_1 \oplus \cdots \oplus x_k)$ out of 2^r many. While this might not be a big issue for CDCL solvers (since corresponding values could be propagated), this is particularly difficult for the probabilistic approach taken by SLS solvers.

Hence, after the XOR constraint becomes satisfied on a high level, the probability of an SLS solver flipping the correct auxiliary variables and satisfying the low-level Tseitin-encoded clauses is small. Worse still, it may end up flipping one of the original variables x_i and invalidating the XOR constraint.

Another issue with the Tseitin encoding of XOR constraints that particularly affects SLS solvers is the change in neighbourhood of assignments within the search space. If we look at an XOR constraint $(x_1 \oplus \cdots \oplus x_k)$ that just got falsified under a certain assignment by flipping x_1, we could easily fix this by flipping an arbitrary variable in this constraint using a single step, including x_k. This does not hold for the Tseitin-encoded CNF version of the XOR constraint anymore. Once again, consider `XOR_1_3`: If the CNF version of the constraint is falsified due to x_1, this can only be fixed by x_2 or y_1. In order to flip x_k, we first would have to take r intermediate steps by flipping all $y_1, \ldots, y_r$. This can be particularly problematic with probabilistic algorithms, considering that the correct variable has to be chosen in each step, decreasing the overall probability for x_k being reached exponentially. Similar arguments have been made in the context of configuration checking [32] and might have contributed to the success of CCA-based solvers, such as `CCAnr` [11].

It is possible to trade off some auxiliary variables for an increased clause count by increasing the cutting number. However, as the number of clauses grows exponentially, this explodes quickly. Using pooled mode, the exponential decrease in probability can be avoided since the resulting structure will roughly be tree-like, i.e., $O(\log(r))$ steps are sufficient for possibly reaching any other variable from the original XOR constraint. Nevertheless, the overall probability distribution is still skewed heavily towards "close" variables and the high-level vs low-level satisfiability issue is not resolved either. Thus, we can only expect small improvements compared to the linear version.

XNF Format

Overall, the currently widely-used Tseitin encoding of XOR constraints is ineffective in SLS solvers. Thus, our goal is to avoid this conversion by including XOR constraints natively as part of the input format. This would enable an SLS solver to handle XOR constraints more effectively, and we hope to standardize this format to facilitate research on SAT solvers with XOR reasoning.

We use the following extension of the existing CNF format that is compatible with XOR constraints. For simplicity, we will call this format XNF. This format is in the spirit of the DIMACS format, which makes it natural to standardize. An XOR constraint is denoted as a sequence of literals preceded by the symbol `x`; OR constraints are denoted the same as in the original CNF format. The

header is changed slightly to "`p xnf #variables #constraints`". For example, the formula $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \oplus x_2)$ is denoted by the following XNF input:

```
p xnf 3 2
1 2 -3 0
x -1 2 0
```

While we came up with this format independently, we later stumbled upon a blog post[4] that briefly mentions CryptoMiniSAT's [38] support of a very similar input format[5]. Support for inputs in XNF also was recently added to the CDCL solver Satch[6], which is CNF-based and uses the pooled encoding presented in Section 2 to encode XOR constraints into CNF. This extension to Satch turned out very useful for testing the tools discussed in the next section.

3 Extracting XORs

Existing propositional problems do have XOR constraints but are usually only available in CNF. Therefore, we have implemented a stand-alone extraction tool `cnf2xnf` which allows to extract an XNF file from a given CNF in DIMACS format. We implemented this tool to make sure that our approach for hybrid local search also works, in a practical sense, with benchmarks given in CNF. The more general goal of this tool is to promote the XNF format and thus further encourage research into hybrid XNF solving.

The SAT solver CryptoMiniSAT [38] contains an internal procedure for extracting XORs [37] in order to take advantage of sophisticated XOR reasoning [35] for applications in approximate model counting [13]. The aim of that extractor is to recover XORs after encoding them into CNF and running CNF-based inprocessing. It takes shortened clauses into account—a common side-effect of CNF-level preprocessing. In earlier work by the fourth author [19], XORs were found by sorting the CNF, which fails to extract XORs with shortened clauses. Our new extractor `cnf2xnf`[7] shares the same problem for preprocessed formulas, but otherwise follows the same principles as used by Soos and Meel [37], apart from not using Bloom filters. In addition to extracting directly encoded XORs, our tool also finds XORs encoded in a Tseitin encoding of And-Inverter-Graphs (AIGs) [28]. Our algorithm is simpler and has successfully been used for gate-extraction to improve bounded variable elimination [15] and to implement Gaussian-elimination in some of the last author's SAT solvers [7,9].

Our extraction algorithm works as follows. We go over all clauses (including binary clauses) and as soon as we find a clause of length k with at most one positive literal, called "base clause", we check whether we can find all 2^{k-1} target clauses obtained from the base clause by negating an even number of

[4] https://www.msoos.org/xor-clauses/
[5] We are not aware of any formal publication about this format.
[6] https://github.com/arminbiere/satch
[7] https://github.com/arminbiere/cnf2xnf

literals. As in subsumption algorithms [15], we only traverse the occurrence list of a single literal in a target clause with the smallest number of occurrences. If all clauses are found, they are marked as garbage and the corresponding XOR constraint is added. Extracting ternary XORs from AIGs starts with a ternary base clause which, together with two binary clauses, encodes an AND gate. For each of the two inputs of that outer AND gate, we then try to find another three clauses encoding an inner AND gate, which share the same inputs but negated. The implied XOR constraint is extracted. If the variables encoding the output of the two inner AND gates occur exclusively in these nine clauses, the clauses are then marked as garbage.

After extracting all XOR constraints, we eliminate variables which only occur in XOR constraints through substitution, simulating Gaussian elimination. The resulting XNF is written to the output file. Optionally, the user can request to produce an "extension stack", listing all the eliminated XOR constraints as well as those sets of nine clauses for XORs extracted from AIGs. This extension stack can be used to map a satisfying assignment of the XNF back to the original CNF. This is implemented in another tool called `extor`[8]. It takes a satisfying assignment of the XNF in the output format of the SAT competition together with the extension stack as inputs and produces a satisfying assignment for the original CNF—again in the SAT competition output format. The algorithm is exactly the same as for reconstructing solutions for CNF preprocessing [16,25,26], except for the semantics of XOR constraints: for those, the value of the first literal of a processed constraint on the stack is flipped if it has an even number of true literals. For regular clauses, the value is only flipped if all literals are false.

As mentioned above, XNF parsing was also added to the new SAT solver Satch, which was then used to test the `cnf2xnf` extractor as well as solution reconstruction with `extor`. For 235 benchmarks of the main track of the SAT competition 2020, we were able to find and extract XORs successfully. From those, a subset of 118 allowed to eliminate variables by Gaussian elimination. This reduced the number of variables substantially—often to less than 50%. However, note that extracting binary XORs partially simulates equivalent literal substitution. Thus, it is difficult to give a precise account of the effectiveness of this flow as a preprocessing technique, which is available in other SAT solvers anyhow and not the target of this paper. Without any bounds, running XOR extraction until completion was able to extract all XORs of 224 benchmarks within one second and all XORs of 333 benchmarks within 10 seconds. For only 30 benchmarks, it took more than 100 seconds.

While these experiments are successful in showing that XNF extraction is feasible on standard competition instances, running Satch on the extracted XNF benchmarks had almost identical performance to running it on the original CNF versions. Furthermore, none of the satisfiable benchmarks was solved through local search, neither before nor after XNF extraction—however, the focus of this paper is to improve local search on specific benchmarks where local search

[8] also available at https://github.com/arminbiere/cnf2xnf

already has an advantage. We consider it a challenge and future work to improve XOR-based reasoning on competition benchmarks.

4 Implementation

To support the XNF format which natively encodes XOR constraints, we modified `YalSAT` [8], a state-of-the-art SLS solver. We call our modified solver `xnfSAT`.[9]

Recall the structure of the `YalSAT` algorithm as outlined in Algorithm 1. Most of the modifications are natural analogies to XOR constraints. The bulk of our modifications concerns the internal data structures and the implementation of *pickVar* (line 8). To perform preprocessing efficiently, we adapted this step to carry out unit propagation on XOR constraints. We did not significantly change *pickUnsatClause* (line 7), as the existing code was sufficient to handle the newly added XOR constraints. For formulas that are encoded in pure CNF, our modification does not change the behavior of `YalSAT`.

For preprocessing, we carry out two rounds of unit propagation on clauses as `YalSAT` does, including also unitary XOR constraints. After unit propagation terminates, we want to utilize the partial assignment forced by unit propagation on XOR constraints. To deduce contradiction is easy, by examining whether there is a falsified XOR constraint. However, to remove satisfied literals, an XOR constraint should have its parity flipped: initially, an XOR constraint is satisfied iff an *odd* number of its literals are set to true; if one of its literals is forced to true by unit propagation, then the XOR constraint is true iff an *even* number of its remaining literals are set to true. As a result, we need an array to keep track of the parity of each XOR constraint.

Define `parity` of an XOR to be 0 if the constraint is satisfied when an odd number of its literals are set to true, and define `parity` to be 1 otherwise. This definition has the convenient property that it is precisely the "base truth value" of the XOR, so that the actual truth value of the constraint in a local search step can be calculated by comparing its current value to its `parity`. Using this definition, we only need to store the variables (but not whether they are negated) appearing in each XOR constraint and initialize its `parity` to the number of negations modulo 2.

Next to basically being a `WalkSAT`-based algorithm, `YalSAT`, more specifically, is also a `probSAT`-based algorithm. In `probSAT` [3], the probability that a variable x is picked is proportional to $c_b^{-break(x)}$, where c_b is a constant, called the *break coefficient*, and $break(x)$ denotes the number of clauses that would be falsified when x was flipped. A key extension of `YalSAT` compared to `probSAT` is that it uses a weighted version of break instead. In `YalSAT` [8], the probability of choosing a variable x is proportional to $c_b^{-break_w(x)}$, with $break_w(x) = \sum_{C \in B(x)} w(C)$, where $B(x)$ is the set of clauses that would be falsified by flipping x, and $w(C)$ is the weight of a particular clause C. In its current implementation within `YalSAT`,

[9] https://github.com/Vtec234/xnfsat

$w(C)$ is not specific to each single clause though, but defined as a function of its length—we will get back to that later.

For `xnfSAT`, we first extend the definition of *break* and $break_w$ by also taking into account the XORs that would be falsified. This is straightforward from a theoretic perspective, but requires to address the concrete implementation as part of an efficient SLS solver architecture. In the original `YalSAT` [8], calculating $break_w$ values using critical literals is crucial to its performance. A literal in a clause C is *critical* if flipping it falsifies C. Say a clause is k-satisfied if k literals in this clause are set to true. Then a clause contains a critical literal iff it is 1-satisfied. Since $break(x)$ is equal to the number of clauses in which x is critical, $break(x)$ can be cached and updated efficiently by tracking the number of true literals in each clause. Whenever a literal is flipped, this can be efficiently updated while looping through each clause where the corresponding variable occurs [1]. This is also where weighting is addressed when implementing $break_w$ in `YalSAT` [8]—instead of just increasing or decreasing the cached value by 1, it can be increased or decreased by $w(C)$, respectively. To generalize this idea to XORs, note that each time a literal in an XOR constraint is flipped, the truth value of the XOR constraint changes. Thus, in a satisfied XOR constraint, every literal is critical. When an XOR constraint C becomes satisfied or unsatisfied, increase or decrease $break_w(x)$ by $w(C)$, respectively, for all its literals x.

In `YalSAT` [8], the weight $w(C)$ is a function of the length of the clause C. However, this is not necessarily a good heuristic to measure the importance of an XOR compared to a clause, especially when XOR constraints are significantly longer. For example, in `MM-Challenge-1`, all XORs are more than six times longer than all the clauses. To simplify the algorithm and not to overtune on specific parameters, we assign a fixed weight w_X to all XOR constraints. Similarly, we will write w_k for $w(C)$ when C is a clause of length k.

Finally, a good choice for c_b is very important and has been extensively studied in the context of distribution-based SLS solvers [1,2,3]—however, mainly on random $k-$SAT problems. With hard combinatorial formulas usually not having uniform clause lengths, the original `YalSAT` [8] automatically configures c_b as a function of the maximum length of all clauses. This is no longer suitable when XOR constraints are added: For one thing, it is not clear whether the length of native XORs should be considered in the same way as the one of clauses. For another, when translated into CNF, the length of the resulting clauses depends on the encoding. To facilitate a thorough evaluation, we thus decided to re-expose c_b as a parameter in `xnfSAT`.

Now there is one remaining issue with `YalSAT`, which initially was very helpful to show the general usefulness of SLS on the `MM-Challenge-1` benchmarks [21], but would prevent a clear analysis of the contribution of native XORs to the algorithm. In its original version [8], `YalSAT` changes *strategy* after each restart interval[10]. This can help find good settings for a broad range of instances and thus is supposed to increase overall robustness. On the negative side, it obfuscates what exactly contributes to a successful run by possibly causing hard to

[10] A detailed explanation of strategies in `YalSAT` is out of the scope of this paper.

predict, unknown interactions. In preliminary experiments, we still had strategies switched on. Implementing XOR support in `xnfSAT` with strategies significantly improved performance (cf. Figure 1), but we soon realized that it is hard to tell if this effect was really just because of the XORs and not due to some hidden interaction with a complex strategy—this could then prevent the same approach to work with other solvers. We thus decided to disable all strategies and to instead figure out which were the individual components that contributed to the good performance on `MM-Challenge-1`. As a side effect, the resulting version of `xnfSAT` became much faster. However, note that our goal was not to overtune to a specific benchmark class nor should this be considered our contribution—instead, the aim was to simplify the algorithm. As our results in Table 1 show, adding native XOR support on top of this much simpler, strategy-free version, still significantly improves performance and we can now conjecture that this is indeed due to our hybrid implementation. The changes we made by switching off strategies:

- Caching is always on, i.e. after a restart the algorithm will pick a previous local minimum. (Had a small effect.)
- c_b is now fixed and never modified during run. (Had a medium effect.)
- Weights w_k for different clause lengths are now exposed as a parameter and never modified during run. (Had a large effect.)
- The initial assignment is now always $0 \dots 0$. (Had the largest effect.)

5 Experiments

We benchmark `xnfSAT` on `MM-Challenge-1`. These instances are hard for CDCL, and best known performance on them has been achieved by SLS [21]. We compare several encodings:

- original, handcrafted XNF (before conversion to CNF)
- CNF with linear `XOR_l_n` constraints and cutting number $n \in [3, 8]$
- CNF with pooled `XOR_p_n` constraints and cutting number $n \in [3, 8]$
- reconstructed XNF as extracted from CNF by `cnf2xnf`[11]

Running on different CNF variants allows us to observe the impact of the choice of XOR encoding on performance. Running on both handcrafted and extracted XNF allows us to verify that the `cnf2xnf` outputs perform adequately compared to hand-written formulas. Note that the runtime of `cnf2xnf` on these instances is negligible, around 0.3s per formula. All benchmark formulas involve a significant amount (more than 700) of XORs or their clausal encodings.

Parameter choices are crucial to the performance of SLS solvers. We optimize parameter classes outlined in Section 4: the break coefficient c_b, the weight w_X assigned to XOR constraints, and the weights w_k assigned to clauses of length k.

In preliminary experiments with the strategy-based version described in Section 4, we first searched for optimal values of c_b and w_X on the 4-cut pooled CNF

[11] The CNF encoding was generated by using `XOR_p_4` on the handcrafted XNF.

(recall that it was conjectured optimal [37]) and on the original XNF. On both formulas, we sampled c_b in the range $[1.5, 5.5]$. On XNF, we also sampled w_X in $[2, 8]$. These ranges were observed to contain most acceptable values. On CNF, the break constant was sampled with a step size of 0.25—on XNF, a step size of 1.0 was used for both parameters due to the higher computational resource requirements of the two-dimensional (c_b, w_X) grid. We found that the average best-performing c_b value for all instances is around 2.5. Interestingly, this does not change with the addition of XOR constraints. The best-performing w_X is around 5.0. The strategy-based versions will not be discussed in detail, but the runtime CDF of the best configuration (with $c_b = 2.5, w_X = 5.0$) on CNF and XNF is plotted in Figure 1 for comparison.

For our full experiments, we then switched off strategies and fixed $c_b = 2.5$ as well as $w_X = 5.0$, next sampling w_k for $k \in [2, 8]$ (there are no clauses of other lengths) on every variant of the instances. The sampled ranges varied as we analyzed preliminary experiments but tended to be within $[2, 5]$. This roughly corresponds to the range that was previously used by the strategies in the original `YalSAT`. However, these values are now fixed and do not change after each restart, making the solver much simpler. Having sampled a broad range of values for w_k, we decided to go for $w_2 = w_3 = 2$, $w_4 = w_5 = 4.5$, and $w_6 = w_7 = w_8 = 5.0$ for all encodings, aside from the 3-cut one, where we chose $w_3 = 4.5$ for reasons that we will explain later. While this setting was not necessarily optimal for each instance, the overall results were solid[12]—recall that our goal was not to perfectly tune every single formula, but to show that the underlying algorithm profits from adding native XOR support. Note that we invested significant computational resources into optimizing CNF weights in order to ensure that our results persist even against well-tuned CNF encodings.

We ran all benchmarks on the Lonestar 5 cluster of Texas at Texas Advanced Computing Center, which has Xeon E5-2690 processors with 24 hardware threads per node. Each variant of each instance was attempted $8 \times 24 = 192$ times (for 192 *runs*) with a timeout of 1000s. Performance is measured by three metrics: the percentage of instances solved within our timeout, the average number of variable megaflips (flips$\times 10^6$) before reaching the solution, and the average time to solution (in seconds). In Table 1, these are abbreviated by `frac`, `Mflips` and `time`, respectively. In addition, to gain more insights into how the specific encoding of XOR constraints impacts performance, we measured the percentage of flips spent on auxiliary variables (`aux`) for the CNF instances.

Results

Figure 1 shows the overall results of our experiments, plotting a runtime CDF of what we consider to be the most interesting configurations. For each encoding, we show the configuration (i.e., choice of parameters) that performed best regarding the overall number of solved instances with that encoding. In general, hardness of the individual instances did not differ a lot among the various encodings and

[12] In the final version, we will replace this note by a link to the full experimental data.

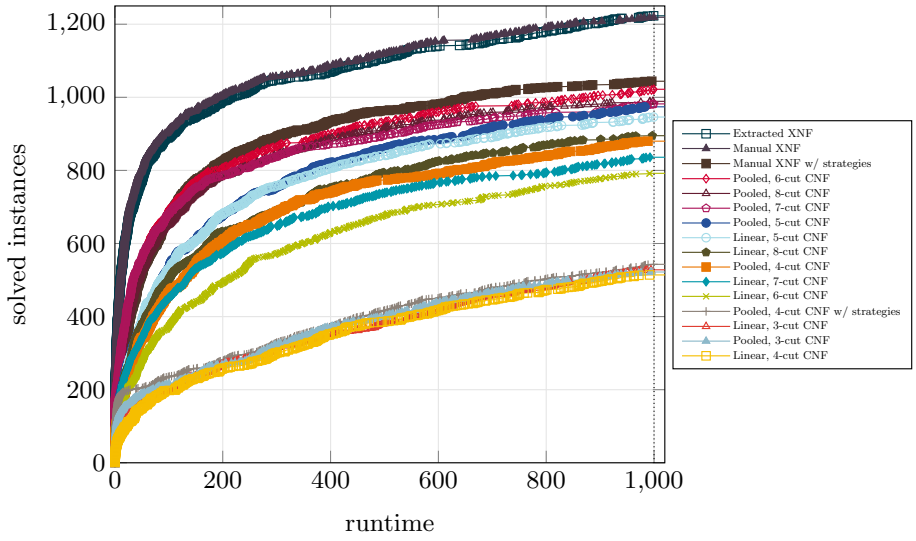

Fig. 1. Runtime CDF of `xnfSAT` performance on various encodings and solver versions.

parameter configurations, i.e., easy instances or hard instances for one setting were also easy or hard, respectively, for all other settings.

We can see that the XNF-based solver outperforms all CNF versions by a huge margin for both the handcrafted encoding as well as the reconstructed XNF. In particular, it takes only 200 seconds for the XNF version to solve approximately the same number of instances as the best CNF configuration.

Furthermore, both XNF versions perform roughly equally—this confirms that `cnf2xnf` was successful in extracting the XORs and shows that the resulting structure is not negatively affected in any way.

Next, we can take a closer look at the different CNF encodings. One trend seems to be that performance on `MM-Challenge-1` increases with larger cutting numbers, reaching peak at 6. In particular, the 4-cut encoding that we initially conjectured to be optimal turned out to perform worse than higher cutting numbers, with just the 3-cut encoding being worse. Note that the 4-cut encoding was used for initially creating the publicly available CNF representation of the `MM-Challenge-1` benchmark set and for first solving it using `YalSAT` [21]. This points towards another central benefit of our native XOR representation, which we have not explicitly discussed so far: while the optimal cutting number turned out to be 6, we do not need to care about finding that out since XNF still outperforms it.

Regarding CNF encodings, there is some mild evidence that the proposed pooled mode is generally better than the linear mode. While this is not true for the 3-cut encoding, we think that this might be a special case that could also be influenced by other factors. Notably, the original XNFs only contain clauses of length 2 and 3. After encoding the instances into CNF, new clauses of length n

Table 1. Performance of `xnfSAT` on benchmarked instances at the optimal settings: CNF ($c_b = 2.5, w_2 = w_3 = 2, w_6 = 5$) and XNF ($c_b = 2.5, w_2 = w_3 = 2, w_X = 5$).

MM-23-*	frac	aux	Mflips	time	frac	aux	Mflips	time	frac	Mflips	time
4-4-4-4-1	76.6	7.8	280.8	67.4	99.5	19.8	31.3	8.2	100.0	0.4	0.1
2-2-2-2-A	94.3	4.1	602.6	154.6	100.0	12.0	245.3	72.7	100.0	55.9	15.6
2-2-2-2-D	79.2	3.2	1171.8	299.6	99.0	10.0	345.7	105.7	100.0	77.1	22.0
2-2-2-3-4	85.4	3.8	967.1	234.6	99.5	11.1	430.0	122.3	100.0	269.0	73.1
2-2-2-2-C	60.4	3.4	1174.1	297.6	85.4	9.7	652.9	192.8	98.4	332.6	90.9
2-2-2-4-B	12.5	4.5	2020.1	487.9	30.7	11.2	1748.8	515.0	42.7	1648.9	432.6
2-2-2-2-B	2.1	4.8	2703.6	676.0	12.5	13.3	1756.5	529.5	41.1	1574.6	429.7
2-2-2-2-M	0.5	3.1	1182.2	316.3	0	–	–	–	29.2	1516.4	450.3
2-2-2-2-3	1.6	3.8	1543.5	385.6	5.7	11.3	2118.0	612.8	23.4	1439.7	392.3
2-2-2-4-A	0	–	–	–	0	–	–	–	2.1	2943.1	835.0
Formula	Linear, 6-cut CNF				Pooled, 6-cut CNF				Extracted XNF		

will be introduced for the n-cut encoding. As a result, the 3-cut encoding is the only one having to use the same weight for original 3-clauses as well as for clauses representing CNF-encoded XORs. Yet again, this points to another benefit of using native XORs: for formulas other than in `MM-Challenge-1`, it may well be the case that clauses in the original XNF have lengths above 3. Thus, even for n-cut encodings with $n > 3$, there is no guarantee that the clauses representing CNF-encoded XORs can be weighted differently from the original clauses.

Looking at Table 1, we can see a detailed analysis of the best-performing linear CNF, pooled CNF, and the XNF version. We chose to display the extracted XNF version because in problems without handcrafted XNF, this could still be obtained using `cnf2xnf`. The XNF version outperforms both CNF encodings across all benchmarks.

6 Conclusion

Combining clausal and XOR reasoning has frequently been looked at in the past. However, rarely so in the context of SLS solvers. With many possible applications, particularly in the domain of cryptography [4,5,18,31] or for approximate model counting [37], progress in this area is certainly of interest.

We argued why CNF encodings of XOR constraints can hinder the performance of SLS solvers and, next to presenting the pooled CNF encoding, advocate for a hybrid representation that allows clauses as well as native XOR constraints. We thus proposed to standardize a format that we call XNF, being a natural extension of the CNF DIMACS format, in order to further support research in that direction. To enable broader use, we also developed the tools `cnf2xnf` and `extor` to find and extract XOR constraints in CNF formulas, convert them into XNF, and to allow reconstructing the solution for the original formula afterwards. We then proceeded by presenting our main contribution, a hybrid SLS solver

called `xnfSAT`. Our detailed experimental evaluation on the matrix multiplication challenge benchmarks [21] showed that xnfSAT solves XNF representations way faster than the corresponding CNF representations, thereby confirming the benefit of supporting native XOR constraints and pushing the state-of-the-art on these instances. As further side results, we presented several other evaluations, providing insights into possible effects that various different CNF encodings as well as parameter settings might have on the performance of SLS solvers.

We hope that our contributions further spark community interest in hybrid SAT solving for clauses and XORs, and expect our results to generalize to other instances with XOR constraints. There are certainly many possible directions of relevance, some of those related to the present work:

While `xnfSAT` implements support for various clause selection heuristics [1], we sticked to the default setting, using *unfair breadth first search* during our evaluation. Note that all those clause selection heuristics were originally developed for pure CNF-based solvers. Nevertheless, preliminary experiments showed that changing the clause selection heuristic can affect the performance of our solver. For future work, it might be interesting to look in more detail at new heuristics which allow treating clauses and XORs in a different manner.

We also noted that using a starting assignment of $0 \dots 0$ was important and performed much better than random initialization for the benchmarks we considered. However, this does not necessarily mean that $0 \dots 0$ is already optimal. Besides, other problem classes could benefit from different initial assignments. Thus, another interesting direction of research could go into the direction of combining the approach used by `NLocalSAT` [41] with `xnfSAT` or with hybrid representations in general.

As we saw in Section 5, the CNF encoding using a cutting number of 3 performed worse compared to the other CNF representations. One reason might be due to the fact that the formulas originally already contain clauses of length 3, but also the XOR constraints are mapped to clauses of this length. While it is likely that the two kinds of clauses should be treated differently, both are assigned the same weight w_3. To address this, individual clause weighting heuristics as part of other solvers [2,39] could be of use. Beyond that, more sophisticated approaches [36] could potentially also be adapted to find individual weights.

Finally, it would also be interesting to look at whether pure CNF-based CDCL solvers can profit from different encodings of XOR constraints, e.g., using the pooled mode in contrast to a standard linear encoding.

Acknowledgements

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. The work is also supported by the National Science Foundation (NSF) under grant CCF-2010951, Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), and the LIT AI Lab funded by the State of Upper Austria.

References

1. Balint, A., Biere, A., Fröhlich, A., Schöning, U.: Improving implementation of SLS solvers for SAT and new heuristics for k-sat with long clauses. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8561, pp. 302–316. Springer (2014)
2. Balint, A., Fröhlich, A.: Improving stochastic local search for SAT with a new probability distribution. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing – SAT 2010. pp. 10–15. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
3. Balint, A., Schöning, U.: Choosing probability distributions for stochastic local search and the role of make versus break. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7317, pp. 16–29. Springer (2012)
4. Bard, G.V.: Introduction: How to Use this Book, pp. 1–6. Springer US, Boston, MA (2009)
5. Bard, G.V., Courtois, N.T., Jefferson., C.: Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF(2) via SAT-solvers. Cryptology ePrint Archive, Report 2007/024 (2007), https://eprint.iacr.org/2007/024
6. Belov, A., Järvisalo, M., Stachniak, Z.: Depth-driven circuit-level stochastic local search for SAT. pp. 504–509 (01 2011)
7. Biere, A.: Lingeling and friends entering the SAT Challenge 2012. In: Balint, A., Belov, A., Diepold, D., Gerber, S., Järvisalo, M., Sinz, C. (eds.) Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2012-2, pp. 33–34. University of Helsinki (2012)
8. Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In: Balyo, T., Heule, M., Järvisalo, M. (eds.) Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2017-1, pp. 14–15. University of Helsinki (2017)
9. Biere, A.: CaDiCaL at the SAT Race 2019. In: Heule, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Race 2019 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2019-1, pp. 8–9. University of Helsinki (2019)
10. Bulygin, S., Buchmann, J.: Algebraic cryptanalysis of the round-reduced and side channel analysis of the full printcipher-48. In: Lin, D., Tsudik, G., Wang, X. (eds.) Cryptology and Network Security. pp. 54–75. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
11. Cai, S., Luo, C., Su, K.: Ccanr: A configuration checking based local search solver for non-random satisfiability. In: Heule, M., Weaver, S. (eds.) Theory and Applications of Satisfiability Testing – SAT 2015. pp. 1–8. Springer International Publishing, Cham (2015)
12. Cai, S., Su, K.: Local search for boolean satisfiability with configuration checking and subscore. Artif. Intell. **204**, 75–98 (2013)

13. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable approximate model counter. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming. pp. 200–216. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
14. Chen, J.: Building a hybrid SAT solver via conflict-driven, look-ahead and XOR reasoning techniques. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5584, pp. 298–311. Springer (2009)
15. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3569, pp. 61–75. Springer (2005)
16. Fazekas, K., Biere, A., Scholl, C.: Incremental inprocessing in SAT solving. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11628, pp. 136–154. Springer (2019)
17. Fröhlich, A., Biere, A., Wintersteiger, C.M., Hamadi, Y.: Stochastic local search for Satisfiability Modulo Theories. In: Proceedings of AAAI. AAAI (January 2015), https://www.microsoft.com/en-us/research/publication/stochastic-local-search-for-satisfiability-modulo-theories/
18. Gwynne, M., Kullmann, O.: On SAT representations of XOR constraints. In: Dediu, A., Martín-Vide, C., Sierra-Rodríguez, J.L., Truthe, B. (eds.) Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8370, pp. 409–420. Springer (2014)
19. Heule, M.J.H.: SmArT solving: tools and techniques for satisfiability solvers. Ph.D. thesis, Delft University of Technology, Netherlands (2008), http://resolver.tudelft.nl/uuid:d41522e3-690a-4eb7-a352-652d39d7ac81
20. Heule, M., van Maaren, H.: Aligning cnf- and equivalence-reasoning. p. 145–156. SAT'04, Springer-Verlag, Berlin, Heidelberg (2004)
21. Heule, M.J.H., Kauers, M., Seidl, M.: Local search for fast matrix multiplication. CoRR **abs/1903.11391** (2019), http://arxiv.org/abs/1903.11391
22. Heule, M.J.H., Kauers, M., Seidl, M.: New ways to multiply 3×3-matrices. J. Symb. Comput. **104**, 899–916 (2021). https://doi.org/10.1016/j.jsc.2020.10.003, https://doi.org/10.1016/j.jsc.2020.10.003
23. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. CoRR **abs/1605.00723** (2016), http://arxiv.org/abs/1605.00723
24. Ishtaiwi, A., Thornton, J., Sattar, A., Pham, D.N.: Neighbourhood clause weight redistribution in local search for sat. In: van Beek, P. (ed.) Principles and Practice of Constraint Programming - CP 2005. pp. 772–776. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
25. Järvisalo, M., Biere, A.: Reconstructing solutions after blocked clause elimination. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6175, pp. 340–345. Springer (2010)

26. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7364, pp. 355–370. Springer (2012)
27. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability. Addison-Wesley Professional, 1st edn. (2015)
28. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust boolean reasoning for equivalence checking and functional property verification. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **21**(12), 1377–1394 (2002)
29. Kyrillidis, A., Shrivastava, A., Vardi, M.Y., Zhang, Z.: Fouriersat: A fourier expansion-based algebraic framework for solving hybrid boolean constraints (2020)
30. Kyrillidis, A., Vardi, M.Y., Zhang, Z.: On continuous local BDD-based search for hybrid SAT solving (2020)
31. Leventi-Peetz, A., Zendel, O., Lennartz, W., Weber, K.: CryptoMiniSat switches-optimization for solving cryptographic instances. In: Berre, D.L., Järvisalo, M. (eds.) Proceedings of Pragmatics of SAT 2015 and 2018. EPiC Series in Computing, vol. 59, pp. 79–93. EasyChair (2019), https://easychair.org/publications/paper/5g6S
32. Luo, C., Cai, S., Wu, W., Su, K.: Double configuration checking in stochastic local search for satisfiability. pp. 2703–2709 (01 2014)
33. Pham, D.N., Thornton, J., Sattar, A.: Building structure into local search for SAT. In: Veloso, M.M. (ed.) IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. pp. 2359–2364 (2007), http://ijcai.org/Proceedings/07/Papers/380.pdf
34. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. Cliques, Coloring, and Satisfiability DIMACS Series in Discrete Mathematics and Theoretical Computer Science p. 521–531 (1996). https://doi.org/10.1090/dimacs/026/25
35. Soos, M., Gocht, S., Meel, K.S.: Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp. 463–484. Springer (2020)
36. Soos, M., Kulkarni, R., Meel, K.S.: Crystalball: Gazing in the black box of SAT solving. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11628, pp. 371–387. Springer (2019)
37. Soos, M., Meel, K.S.: BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In: AAAI. pp. 1592–1599. AAAI Press (2019), http://dblp.uni-trier.de/db/conf/aaai/aaai2019.html#SoosM19
38. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5584, pp. 244–257. Springer (2009)
39. Thornton, J., Pham, D.N., Bain, S., Jr., V.F.: Additive versus multiplicative clause weighting for SAT. In: McGuinness, D.L., Ferguson, G. (eds.) Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference

on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA. pp. 191–196. AAAI Press / The MIT Press (2004), http://www.aaai.org/Library/AAAI/2004/aaai04-031.php
40. Tseitin, G.S.: On the complexity of derivation in propositional calculus. Automation of Reasoning p. 466–483 (1983)
41. Zhang, W., Sun, Z., Zhu, Q., Li, G., Cai, S., Xiong, Y., Zhang, L.: NLocalSAT: Boosting local search with solution prediction. Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (Jul 2020)