Logical Methods in Computer Science
Volume 22, Issue 2, 2026, pp. 23:1–23:22
https://lmcs.episciences.org/

Submitted Jun. 25, 2025
Published May 25, 2026

EXTENDED RESOLUTION CLAUSE LEARNING VIA DUAL IMPLICATION POINTS

SAM BUSS [a], JONATHAN CHUNG [b], VIJAY GANESH [c], AND ALBERT OLIVERAS [d]

[a] University of California, San Diego, La Jolla, CA, USA

[b] Lorica Cybersecurity, Canada

[c] Georgia Institute of Technology, Atlanta, USA

[d] Universitat Politècnica de Catalunya, Barcelona

Abstract. We present a new extended resolution clause learning (ERCL) algorithm, implemented as part of a conflict-driven clause-learning (CDCL) SAT solver, wherein new variables are dynamically introduced as definitions for *Dual Implication Points* (DIPs) in the implication graph constructed by the solver at runtime. DIPs are generalizations of unique implication points and can be informally viewed as a pair of dominator nodes, from the decision variable at the highest decision level to the conflict node, in an implication graph. We perform extensive experimental evaluation to establish the efficacy of our ERCL method, implemented as part of the MapleLCM SAT solver and dubbed xMapleLCM, against several leading solvers such as Kissat, CryptoMiniSat, and SBVA-CaDiCaL, the winner of SAT Competition 2023. We show that xMapleLCM outperforms these solvers on Tseitin, XORified formulas and Interval Matching problems. We further compare xMapleLCM with GlucosER, a system that implements extended resolution in a different way, and provide a detailed comparative analysis of their performance.

1. Introduction

Over the last several years, Conflict-Driven Clause-Learning (CDCL) SAT solvers have had a dramatic impact on many fields including software engineering [Kro21], security [XA05], and AI [KS92, Kau06]. As solvers continue to be adopted in increasing complex settings, the demand for greater efficiency and reasoning power by users continues unabated.

While developers continue to improve CDCL SAT solvers, it is simultaneously true that the base solvers are provably no more powerful than the relatively weak general resolution (Res) proof system [AFT11a, PD10], and therefore are fundamentally limited. Hence, solver developers have been actively researching novel algorithms that implement stronger proof systems that go beyond Res. Examples of such algorithms include satisfaction-driven clause-learning (SDCL) SAT solvers [HKB19, OLW$^+$23], bounded variable addition

Key words and phrases: SAT, Extended resolution, Conflict Analysis.
Albert Oliveras is supported by grant PID2021-122830OB-C43, funded by MCIN/AEI/10.13039/501100011033 and by "ERDF: A way of making Europe"; by grant PID2024-157044OB-C32, funded by MICIU /AEI /10.13039/501100011033 / FEDER, UE. and by Barcelogic through research grant C-11423.

DOI:10.46298/LMCS-22(2:23)2026

(BVA) [MHB12], symmetry breaking [Sak21], and extended resolution (ER) solvers such as GlucosER [AKS10].

Continuing this trend of strong proof system implementations, we present a new extended resolution clause learning (ERCL) algorithm, incorporated into a CDCL SAT solver, where new variables are dynamically introduced as definitions for *dual implication points* (DIPs) in the implication graph constructed by the solver at run time. The concept of a DIP is best understood as a generalization of unique implication points (UIPs). Informally, a UIP can be defined as a dominator node in an implication graph, corresponding to a variable at the highest decision level (DL), that *dominates* all paths from the decision variable node at the highest DL to the conflict node. By contrast, a DIP is a pair of dominator nodes in an implication graph such that any path from the decision variable node at the highest DL to the conflict node must pass through at least one node in the pair[1].

Implementation of the ERCL algorithm requires several additional methods. First, we need a method to identify DIPs, i.e., a technique that takes as input an implication graph and outputs a DIP and does so in time linear in the size of the input. Second, we need a technique replaces this DIP pair with a new variable and appropriately modifies the clause learning algorithm to learn new clauses involving DIPs. Third, we need an ER framework, built on top of a CDCL SAT solver, that enables new variable addition, ER clause addition and deletion, etc. Finally, we need heuristics that specialize the above mentioned methods in a variety of ways, such as clause learning and clause deletion policies that are based on different kinds of DIPs. We implement all of these methods as part of the MapleLCM solver [LLX$^+$17], and refer to the resulting solver as xMapleLCM. In fact our proposed ERCL method, and its implementation, is very general and easily extensible thus encouraging future exploration and specialization efforts with a variety of heuristics.

Contributions.

(1) **DIP:** We introduce the concept of dual implication points (DIP), a generalization of UIPs in conflict graphs. We also came up with an algorithm that computes them in linear time. However, due to space limitations and to the non-trivial nature of the procedure, we discuss this in a separate paper [BGO24].

(2) **ERCL Algorithm:** We introduce a highly parameterizable DIP-based ERCL algorithm. The existence of a multitude of different DIPs in a single conflict graph allows us to derive a large variety of ERCL algorithms. This flexibility is crucial in adapting the procedure to different scenarios, unlike previous methods that couple CDCL with extended resolution.

(3) **xMapleLCM**: We present a highly extensible and general ER framework as part of xMapleLCM, which allows developers to easily add their own new variable addition, ER clause learning/deletion, and branching policies. Given that DIP-based ERCL is highly flexible by nature, such a framework is necessary to quickly prototype new procedures.

(4) **Experiments:** We perform extensive empirical evaluation and ablation studies on four different classes of instances, namely, SAT Competition 2023 Main Track, random k-xor, Tseitin, and Interval Matching, and compare xMapleLCM against leading solvers such as Kissat [BFFH20], CryptoMiniSat [SNC09], SBVA-CaDiCaL [HGH23], and the

[1] While it is natural to generalize the concept of a DIP to K-Implication Points or k-IPs, we do not discuss them in this paper.

extended-resolution solver GlucosER [AS09]. Results show that on the last three sets of hard combinatorial formulas, CDCL SAT solvers perform very poorly, whereas both xMapleLCM and GlucosER present much better performance. This demonstrates that extended resolution can be added to CDCL and improve the performance of these solvers; GlucosER and our DIP-based methods give two different ways to achieve this. Moreover, for our DIP-based learning, a simple heuristic allows us to detect on-the-fly whether extended resolution is being helpful and revert to using standard CDCL in order to get the best of both worlds. This technique enables xMapleLCM to perform similarly to MapleLCM on the SAT Competition 2023 Main Track instances.

2. Related Work

The idea of using ER in SAT solving has been studied in various forms in the literature for nearly two decades. The closest approach to ours is GlucosER [AKS10], where extended variables are introduced dynamically during the CDCL search: whenever two consecutive learned lemmas are of the form $\neg l_1 \vee C$ and $\neg l_2 \vee C$, with l_1 and l_2 being their UIPs, then an extended variable $z \leftrightarrow l_1 \vee l_2$ is generated and any future lemma of the form $l_1 \vee l_2 \vee D$ is replaced by $z \vee D$. Hence, information across different conflict analysis steps is used to detect which extended variables to introduce.

Our method differs significantly from GlucosER in the way extended variables are identified: we choose DIPs, which can be seen as pairs of variables for which adding a definition would create a better first UIP (often written as 1UIP) in the conflict graph, whereas GlucosER definitions are constructed using already existing UIPs. Our method focuses on a single conflict analysis step to extract an extended resolution variable, whereas GlucosER uses two steps. Also, unlike in GlucosER, our approach does not always learn the standard 1UIP clause, but multiple clauses might be learned that take into account the newly introduced variable. On the other hand, there are similarities at the heart of both procedures: a certain restriction of ER is considered and introduced variables are used to shorten subsequently found clauses.

Another related procedure is presented in [Hua10]. For a learned clause C of size > 2, the author suggests to split C into $\alpha \vee \beta$, where $|\alpha| \geq 2$ and β is non-empty. Instead of learning C, the solver learns $x \vee \beta$ and $x \leftrightarrow \alpha$, where x is a fresh variable. Experimental results in [Hua10] are limited, and no implementation is available.

More recent uses of ER in SAT solvers are Bounded Variable Addition (BVA) [MHB12] and its structured version SBVA [HGH23], that due to clever strategies are able to identify new extended variables whose introduction can reduce the number of clauses. Essentially, the ultimate goal of BVA techniques is to reduce the size of the formula by reducing the number of clauses at the cost of adding a new variable. One big difference with our work is that this process is done only in a preprocessing step. Finally, one additional direction that has been researched is the development of a BDD-based solver to generate ER proofs [BH23].

Other approaches aiming at improving CDCL solvers by allowing them to use a more powerful proof system are related to the Propagation Redundancy notion [HKB17, HKB20], either via preprocessing steps [RHB22] or via the use of the SDCL algorithm [HKSB17, OLW$^+$23]. While these methods implement proof systems that are stronger than Res, many of them (without new variable addition) are known to be weaker than ER.

Another related work is [PD08], where the authors introduce a new conflict analysis procedure that learns lemmas that contain two literals of the last decision level. These two literals are indeed a DIP. However, there can be many other candidate DIP's that are not detected by the methods of [PD08]. Furthermore, as we will show in Section 4.1, their procedure might sometimes fail to detect any DIPs at all. An additional, important differences w.r.t. our work is that [PD08] do not introduce a new variable representing the conjunction of the DIP members. Hence, this is not an extended resolution-based approach and, from the proof-complexity point of view, is not more powerful than standard CDCL or general resolution.

3. Preliminaries

We assume that the reader is familiar with the satisfiability (SAT) problem and the CDCL algorithm, and we refer her to the Handbook of Satisfiability for an excellent overview of these topics [MLM21]. Below we focus on conflict analysis, which is the most relevant ingredient from the CDCL algorithm for this paper. We do so by means of the following example.

Example 3.1. Consider the following clauses

(1)	$y_1 \vee \neg x_1 \vee x_2$	(6)	$\neg x_5 \vee x_7$	(11)	$\neg x_{11} \vee x_{12}$
(2)	$\neg x_1 \vee \neg x_3$	(7)	$x_6 \vee \neg x_7 \vee x_8$	(12)	$x_{10} \vee \neg x_{11} \vee x_{13}$
(3)	$y_2 \vee \neg x_1 \vee x_4$	(8)	$\neg y_3 \vee \neg y_4 \vee \neg x_5 \vee \neg x_9$	(13)	$x_{12} \vee \neg x_{13}$
(4)	$\neg y_3 \vee \neg x_2 \vee x_3 \vee \neg x_4 \vee x_5$	(9)	$\neg y_4 \vee x_9 \vee \neg x_{10}$		
(5)	$y_1 \vee \neg x_5 \vee \neg x_6$	(10)	$\neg y_5 \vee y_6 \vee \neg x_8 \vee x_9 \vee x_{11}$		

Assume that CDCL has constructed an assignment that contains, among others, literals $\{\neg y_1, \neg y_2, y_3, y_4, y_5, \neg y_6\}$. Since no propagation is possible, it now decides to add the decision literal x_1. Due to clause (1), we can unit propagate literal x_2, being (1) the reason of x_2 and its antecedents $\{\neg y_1, x_1\}$. Similarly, $\neg x_3$ is propagated due to reason (2), with antecedents $\{x_1\}$, and x_4 due to reason (3) with antecedents $\{\neg y_2, x_1\}$. If we continue this process we eventually find that clause (13) is conflicting and we can construct the *conflict graph* in Figure 1, where every literal in the current decision level has incoming edges corresponding to its antecedents (except of course the decision literal). A special conflict node $\perp$ with incoming edges $\{\overline{x_{12}}, x_{13}\}$ represents conflicting clause (13).

The graph clearly shows that if we set x_1 to true, together with the literals of previous decision levels (the y's), we obtain a conflict. However, the same happens with x_5, since any path from x_1 to the conflict necessarily goes through x_5. Literals with this property are called *Unique Implication Points* (UIPs), of which we only have x_1 and x_5. Since x_5 is the one closest to the conflict we call it *First Unique Implication Point* (1UIP) [ZMMM01]. It is easy to see that if we set x_5 and the y literals that enter the cut delimited by the blue line, unit propagation derives the same conflict. Hence, since they cannot be simultaneously true, we can learn $y_1 \vee \neg y_3 \vee \neg y_4 \vee \neg y_5 \vee y_6 \vee \neg x_5$. The quality of a lemma can be assessed by its *Literal Block Distance (LBD)* [AS09]: the number of different decision levels of the literals in the lemma. The lower the LBD, the better the lemma. In our case, if we are at decision level 5, y_3, y_4 and y_6 belong to decision level 2, and y_1 and y_5 to decision level 4, the LBD of the lemma is 3. □

Resolution. Given two clauses $l \vee C$ and $\neg l \vee D$, the *resolution* inference rule allows one to derive the logical consequence $C \vee D$. It is well known that the lemma derived in Example 3.1 can be obtained via a series of resolution steps that start with the conflicting

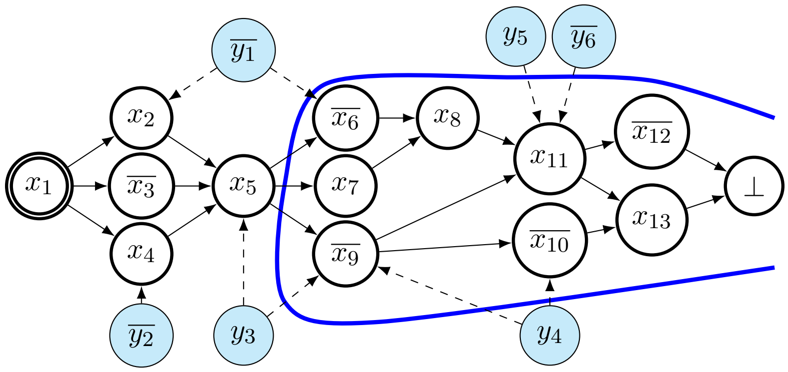


Figure 1: Conflict graph associated with Example 3.1. If 1UIP learning is applied, we generate lemma $y_1 \vee \neg y_3 \vee \neg y_4 \vee \neg y_5 \vee y_6 \vee \neg x_5$. White nodes belong to the current decision level, whereas blue ones are from previous decision levels.

clause, and resolve with reasons of literals in the reverse order in which they were added to the assignment. In fact, it has been proved [PD11, AFT11b] that resolution and CDCL (with restarts) are polynomially equivalent, and hence classes of formulas that are hard for resolution; e.g., the pigeonhole principle - PHP [Hak85], or Tseitin formulas [Urq87]) are also hard for CDCL SAT solvers.

Extended Resolution (ER). Given two literals l_1 and l_2, the *extended resolution* [Tse83] rule allows us to introduce clauses representing the definition $z \leftrightarrow l_1 \vee l_2$. ER can be substantially more powerful than resolution; for instance, it allows polynomial size proofs of PHP [Coo76]. Incorporating ER to CDCL solvers could potentially enable them to solve such formulas in polynomial time. However, we lack good methods to incorporate ER into CDCL proof search. This is precisely the aim of this paper, namely, incorporating a restricted version of ER into CDCL.

4. Dual Implication Points

As discussed in the previous section, UIPs are crucial for conflict clause learning in the CDCL algorithm. We now introduce a new concept of a *Dual Implication Point* (DIP) that gives a tool for analyzing the conflict graph. Its applications include discovering new implied 2-clauses, introducing new variables by extension, and learning clauses involving the extension variables. The idea behind a dual implication point is that it consists of a pair of vertices (literals) in the conflict graph that disconnects or "separates" the decision literal from the contradiction.[2] More precisely, a DIP is defined to be a pair $\{x, y\}$ of literals such that all paths in the conflict graph to the vertex $\perp$ pass through at least one of x and y and such that neither x nor y is a UIP. In contrast, UIP is a *single* literal that separates the decision literal from the conflict.

We use the example in Figure 1 to illustrate the concept of DIPs and their potential applications. Recall that x_5 is the first UIP. In our applications, we are seeking DIPs between the first UIP and the conflict node $\perp$. An obvious DIP is the pair $\overline{x_{10}}$ and x_{11}, since it is

[2]Perhaps "Dual Implication Pair" would be a better name than "Dual Implication Point', since a DIP is a pair of literals. We use "Point" however to match the terminology of "Unique Implication Points".

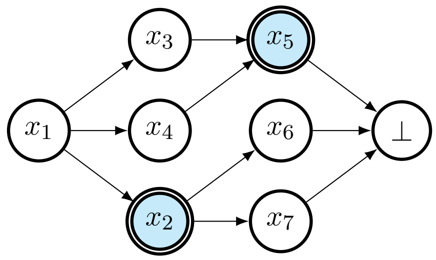
Dual Implication Point(DIP)	Extension Variable	Post-DIP Learned Clause	Pre-DIP Learned Clause
$\overline{x_{12}}, x_{13}$	$z \leftrightarrow (\overline{x_{12}} \wedge x_{13})$	$\neg z$	$\neg x_5 \vee y_1 \vee \neg y_3 \vee \neg y_4 \vee \neg y_5 \vee y_6 \vee z$
x_{11}, x_{13}	$z \leftrightarrow (x_{11} \wedge x_{13})$	$\neg z$	$\neg x_5 \vee y_1 \vee \neg y_3 \vee \neg y_4 \vee \neg y_5 \vee y_6 \vee z$
$\overline{x_{10}}, x_{11}$	$z \leftrightarrow (\overline{x_{10}} \wedge x_{11})$	$\neg z$	$\neg x_5 \vee y_1 \vee \neg y_3 \vee \neg y_4 \vee \neg y_5 \vee y_6 \vee z$
$\overline{x_9}, x_{11}$	$z \leftrightarrow (\overline{x_9} \wedge x_{11})$	$\neg z \vee \neg y_4$	$\neg x_5 \vee y_1 \vee \neg y_3 \vee \neg y_4 \vee \neg y_5 \vee y_6 \vee z$
$x_8, \overline{x_9}$	$z \leftrightarrow (x_8 \wedge \overline{x_9})$	$\neg z \vee \neg y_4 \vee \neg y_5 \vee y_6$	$\neg x_5 \vee y_1 \vee \neg y_3 \vee \neg y_4 \vee z$

Figure 2: The complete list of DIPs for the conflict graph of Figure 1. The DIP-learnable clauses involve the new extension variable z that can be introduced for that DIP. (The extension clauses defining z must also be learned; e.g., for the first line, the extension clauses express that $z \leftrightarrow \overline{x_{12}} \wedge x_{13}$.)

immediately clear that any path from x_5 (or from x_1) must pass through one of $\overline{x_{10}}$ or x_{11}. On the other hand, the pair $\overline{x_{10}}$ and x_8 is not a DIP since there are paths from x_1 to $\perp$ that avoid these two literals; namely, any path that includes the edge from $\overline{x_9}$ to x_{11}. There are several other DIPs in Figure 1: a complete list is given in Figure 2.

Figure 2 also shows how a DIP pair can be used to introduce a variable z via extension, and the associated clauses that can be learned. For example, in the third line, the new extension variable z is introduced with the three clauses $\neg z \vee \neg x_{10}$, $\neg z \vee x_{11}$ and $x_{10} \vee \neg x_{11} \vee \neg z$ which express the condition $z \leftrightarrow (\neg x_{10} \wedge x_{11})$. From the conflict graph, this allows inferring the clauses $\neg z$ (the "post-DIP" learned clause) and $\neg x_5 \vee y_1 \vee \neg y_3 \vee \neg y_4 \vee \neg y_5 \vee y_6 \vee z$ (the "pre-DIP" learned clause), both of which will be formally defined in the next paragraph. Since the post-DIP learned clause does not have any variables from lower decision levels, we can also infer the 2-clause $x_{10} \vee \neg x_{11}$ either instead of or in addition to introducing z and the pre- and post-DIP clauses. Introducing 2-clauses in this way might be helpful for CDCL solvers that do special processing of 2-clauses; for instance, in the work of Bacchus [Bac02] or the recent work of Biere et al. [BFW23] or Buss et al. [BKW24].

In general, for any literals a, b that form a DIP in the above fashion, we may introduce an extension variable $z \leftrightarrow a \wedge b$, and learn (1) a pre-DIP clause of the form $\neg f \vee \neg C \vee z$ (i.e. $f \wedge C \rightarrow z$), where f is the first UIP and C the set of literals from previous decision levels that have an edge in the conflict graph to any literal appearing after the first UIP and no later than the DIP pair; and (2) a post-DIP clause of the form $\neg z \vee \neg D$ (i.e. $z \wedge D \rightarrow \perp$), where D contains those literals from previous decision levels with an edge to any literal appearing strictly after the DIP pair. The correctness of adding these two clauses is guaranteed because they can be generated via a sequence of resolution steps on the formula and the clauses that define the extended variable z. We can also prove their correctness with a more conflict-graph oriented argument: it should be obvious from the conflict graph that if we assert the 1UIP f and all literals in C, we can obtain a and b via unit propagation. This means that $f \wedge C \rightarrow a \wedge b$ is a valid formula which, after replacing $a \wedge b$ by their definition z, becomes the pre-DIP clause $\neg f \vee \neg C \vee z$. Similarly, if we assert a, b and all literals in D, we get a conflict by unit propagation. This means that $\neg a \vee \neg b \vee \neg D$ is a valid clause. If we apply two resolution steps with the clauses $a \vee \neg z$ and $b \vee \neg z$ we obtain the post-DIP clause $\neg z \vee \neg D$.



Trail	Derived clause
$\bot$	$\neg x_5 \lor \neg x_6 \lor \neg x_7$
x_5	$\neg x_3 \lor \neg x_4 \lor \neg x_6 \lor \neg x_7$
x_7	$\neg x_3 \lor \neg x_4 \lor \neg x_6 \lor \neg x_2$
x_6	$\neg x_3 \lor \neg x_4 \lor \neg x_2$
x_4	$\neg x_3 \lor \neg x_1 \lor \neg x_2$
x_3	$\neg x_1 \lor \neg x_2$
x_2	$\neg x_1$
x_1	

Figure 3: Conflict graph associated with Example 4.1, with the literals in blue being the only DIP. Next to it, the resulting trail and the sequence of intermediate clauses obtained if standard 1UIP learning is applied.

For example, the last line of the table in Figure 2 shows the case $z \leftrightarrow (x_8 \land \overline{x_9})$, where f is x_5, and C and D are $\neg y_1 \land y_3 \land y_4$ and $y_4 \land y_5 \land \neg y_6$, respectively. In Section 5.2 we discuss many possible ways that DIP extension variables and clauses may be introduced into CDCL solvers. The rest of this section describes how to find DIPs and its content is not strictly necessary to understand the rest of the paper.

4.1. **An Algorithm for Finding DIPs.** A conventional CDCL algorithm maintains a trail of the literals set true, in the order they were set, and this allows finding the UIP very quickly. The idea is to initialize a clause C as the conflicting clause and, following the order of the trail from the most recent literal l backwards, apply resolution between C and the reason of l to get a new clause C. This is repeated until C only has only one literal of the current decision level.

One could think that finding DIPs can be done by applying the same algorithm and stopping when C has exactly two literals of the current decision level. This is what is done in [PD08] but, as we now show, this procedure is not complete for detecting DIPs.

Example 4.1. Consider the set of clauses $\{\neg x_1 \lor x_2,\ \neg x_1 \lor x_3,\ \neg x_1 \lor x_4,\ \neg x_3 \lor \neg x_4 \lor x_5,\ \neg x_2 \lor x_6,\ \neg x_2 \lor x_7,\ \neg x_5 \lor \neg x_6 \lor \neg x_7\}$. Setting decision x_1 and applying unit propagation finds a conflict. This is illustrated in the conflict graph of Figure 3, where the only DIP is the pair $\{x_2, x_5\}$. In the same figure, we can see the corresponding trail and the intermediate clauses obtained in a standard conflict analysis procedure. Note that the procedure is unable to detect the DIP.

Finding the DIPs is much more complex than finding the UIP, and requires several traversals of the conflict graph between the first-UIP and the conflict node. Nonetheless, it is possible to find all DIPs very quickly, even in linear time.

We recast the DIP-finding problem in terms of a general directed graph G. Let $G = (V, E)$ be a directed graph with two distinguished vertices s and t. We assume that s is 2-connected to t in that there is a pair of vertex-disjoint paths π_1 and π_2 from s to t that share no vertices apart from s and t. By the vertex-version of Menger's theorem [Men27], either there are in fact *three* vertex-disjoint paths from s to t or there is at least one pair of vertices $\{a, b\}$ with neither a nor b in $\{s, t\}$ so that every path from s to t passes through at least one of a or b. Such a pair $\{a, b\}$, if it exists, is called a *Two Vertex Dominator* (TVD).

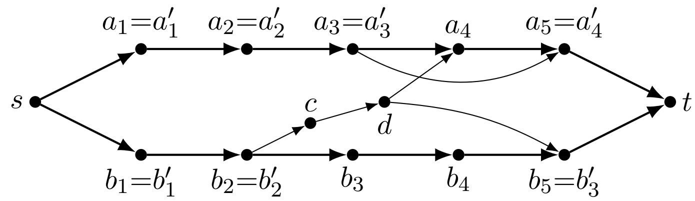


Figure 4: Nodes a_4, b_3 and b_4 are bypassed and thus not in any TVD. The path b_2, c, d, a_4 is a crossing separator that prevents any of a_1, a_2 and a_3 from being paired with b_5 to form a TVD pair. All other pairs $\{a'_i, b'_j\}$ are valid TVD pairs.

Our goal is to find all possible TVDs efficiently and in linear time. An algorithm for this is discussed in detail in our companion paper [BGO24]; for space reasons, we give only an abbreviated discussion of the algorithm here and direct the reader who wants a deep understanding of the algorithm to [BGO24]. The first step is to find two vertex disjoint paths from s to t: this is done by greedily finding a path from s to t and then using an augmenting path[3] construction to find two vertex-disjoint paths from s to t. An example is shown in Figure 4, where the two paths are $a_0, \ldots, a_\ell$ and $b_0, \ldots, b_k$ where $a_0 = b_0 = s$ and $a_\ell = b_k = t$. These paths are called π_a and π_b, respectively. Henceforth, a *path* is a directed path without any repeated nodes. The *internal* vertices of a path π are the vertices on π other than the first and last vertices. Two paths are said to be *vertex-disjoint* if they have no internal vertices in common. A path π *avoids* π_a *and* π_b if it is vertex-disjoint from both paths.

Once the two vertex-disjoint paths are fixed, we can state the following definitions and theorems:

Definition 4.2 [BGO24]**.** A node a_i on π_a is *bypassed* if there are $j < i < j'$ and a path π from a_j to $a_{j'}$ such that π avoids π_a and π_b. A node b_i being bypassed is defined similarly.

Definition 4.3. Two nodes a_i and b_j have a *crossing separator* if there are nodes $a_{i'}$ and $b_{j'}$ joined by a path π that avoids both π_a and π_b such that either (a) $i' < i$ and $j' > j$ and π is a path from $a_{i'}$ to $b_{j'}$, or (b) $i' > i$ and $j' < j$ and π is a path from $b_{j'}$ to $a_{i'}$.

Theorem 4.4 [BGO24]**.** *For $0 < i < \ell$ and $0 < j < k$, the two nodes a_i and b_j form a two-vertex dominator (TVD) if and only if a_i and b_j do not have a crossing separator and neither a_i nor b_j is bypassed.*

Theorem 4.4 is proved in [BGO24]. The theorem holds for both acyclic and cyclic directed graphs; however, for our applications to CDCL we are interested only in acyclic graphs since the conflict graph is always acyclic.

A consequence of Theorem 4.4 is that the set of all TVDs can be compactly represented in linear size, even though there can be quadratically many TVDs. Let $a'_1, \ldots a'_{\ell'}$ be the subsequence of the internal nodes $a_1 \ldots, a_{\ell-1}$ of path π_a that, according to the conditions of Theorem 4.4, are in at least one TVD pair. Let $b'_1, \ldots, b'_{k'}$ be the corresponding subsequence of the internal nodes of π_b. Then, for each a'_i there are $m \leq n$ such that a'_i forms a TVD pair with each b'_j with $m \leq j \leq n$. Dually, for each b'_i there are $m \leq n$ such that b'_i forms a TVD pair with each a'_j with $m \leq j \leq n$.

[3] This can also be computed in linear time by applying Ford-Fulkerson's algorithm to a small modification of the conflict graph in order look for a flow of 2.

Example 4.5. In the conflict graph of Figure 1, consider the portion of the graph between the first UIP x_5 and the contradiction $\bot$. We can take path π_a to be $x_5, x_7, x_8, x_{11}, \overline{x_{12}}, \bot$ and path π_b to be $x_5, \overline{x_9}, \overline{x_{10}}, x_{13}, \bot$. The node x_7 is bypassed by the path $x_5, \overline{x_6}, x_8$, so it cannot be part of a TVD. There are two crossing separator paths: the first is the edge from $\overline{x_9}$ to x_{11}; the second is the edge from x_{11} to x_{13}. Therefore, the possible TVD pairs can be described in a table as:

Node on π_a	forms a TVD pair with	Node on π_b	forms a TVD pair with
x_8	$\overline{x_9}$	$\overline{x_9}$	x_8, x_{11}
x_{11}	$\overline{x_9}, \overline{x}_{10}, x_{13}$	$\overline{x_{10}}$	x_{11}
$\overline{x_{12}}$	x_{13}	x_{13}	$x_{11}, \overline{x}_{12}$

In this example, one node, x_7, was bypassed. It is also possible that non-bypassed nodes are eliminated just by the crossing separators. For example, if there were an additional edge from x_7 to $\overline{x_{10}}$, then the crossing separator condition would imply that x_8 and x_{11} are not part of any TVD pair. In this case, x_8 and x_{11} would not be included among the a'_i nodes.

Theorem 4.4 is used in [BGO24] to give an efficient, linear time algorithm for finding all TVDs. The algorithm has five phases. The first two phases find two vertex-disjoint paths from s to t; the next phase scans the graph from t to s to discover all relevant paths that avoid π_a and π_b; the fourth phase uses this to discard bypassed nodes and collect information on crossing separators; finally the fifth phase computes the compressed representation of all possible TVDs. Full details are given [BGO24], which is available online in preprint form. In our experiments with the implication graphs constructed by the underlying CDCL solver, the time overhead in finding the TVDs is negligible.

5. Extension Variables from Dual Implication Points

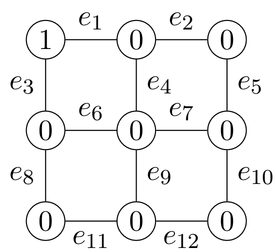
This section discusses how DIPs can be used for introducing extension variables and implement the ERCL method for learning ER clauses in CDCL solver. We first present an example.

5.1. **An example with a grid Tseitin principle.**

Given a graph where every vertex has a *charge*, a number which is 0 or 1, a Tseitin formula [Tse83] is created by considering one variable per edge, and adding one constraint per vertex v expressing that the sum of the variables of all edges incident to v modulo 2 is equal to its charge. The CNF version that we consider converts each *xor* constraint into clauses by simple enumeration of falsifying assignments. It is easy to see that the formula is unsatisfiable if and only if the sum of all charges is odd. Here we consider as a graph the 3×3-grid depicted in Figure 5 and only one vertex has charge 1, so the clauses are unsatisfiable.

The first steps of the CDCL solver are as follows: First e_1 is set true as a decision literal, and $\overline{e_3}$ is (unit) propagated. Second, e_2 is set true as a decision literal, and $\overline{e_4}$ and e_5 are propagated. Third, e_6 is set true as a decision literal, and e_8 and e_{11} are propagated. Fourth, e_7 is set true as a decision literal, and $\overline{e_9}$, $\overline{e_{10}}$, and e_{12} are propagated. This gives a contradiction, since the clause $e_{10} \vee \overline{e_{12}}$ (one of the two clauses from $e_{10} \oplus e_{12} = 0$) is falsified. Figure 6(a) shows the complete conflict graph at this point.

Examining the conflict graph at decision level 4, the first UIP is e_7 and there are two DIPs available, $\{\overline{e_9}, \overline{e_{10}}\}$ and $\{\overline{e_{10}}, e_{12}\}$. Selecting the former DIP, we introduce a new



$$\begin{aligned} e_1 \oplus e_3 &= 1 \\ e_1 \oplus e_2 \oplus e_4 &= 0 \\ e_2 \oplus e_5 &= 0 \\ e_3 \oplus e_6 \oplus e_8 &= 0 \\ e_4 \oplus e_6 \oplus e_7 \oplus e_9 &= 0 \end{aligned} \qquad \begin{aligned} e_5 \oplus e_7 \oplus e_{10} &= 0 \\ e_8 \oplus e_{11} &= 0 \\ e_9 \oplus e_{11} \oplus e_{12} &= 0 \\ e_{10} \oplus e_{12} &= 0 \end{aligned}$$

Figure 5: An instance of the 3×3-grid Tseitin principle. The nodes are assigned a polarity in $\{0, 1\}$. The edges are labeled with variables.

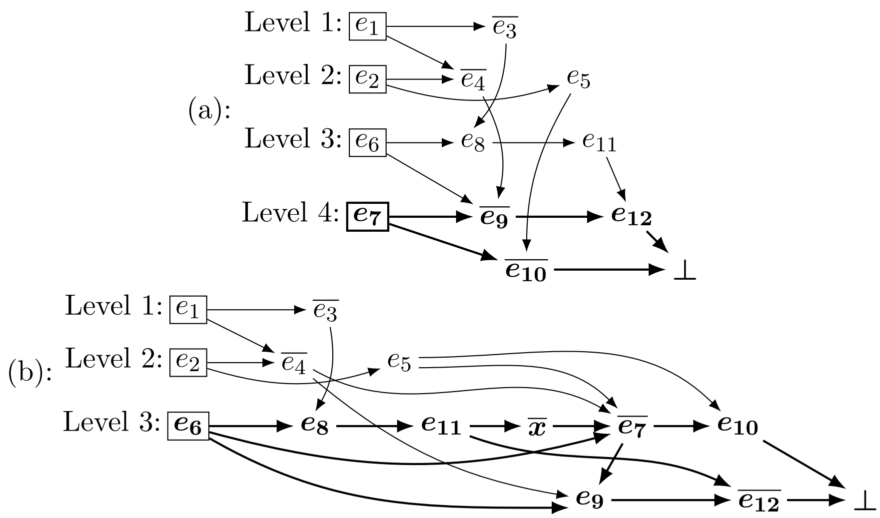


Figure 6: The complete conflict graph from the example; (a) at the first conflict and (b) at the second conflict. Boxed literals are decision literals. Implications at the top decision level are in bold.

variable x by extension as $x \leftrightarrow \overline{e_9} \wedge \overline{e_{10}}$. We can learn the additional pre- and post-DIP clauses: $e_4 \vee \overline{e_5} \vee \overline{e_6} \vee \overline{e_7} \vee x$ and $\overline{x} \vee \overline{e_{11}}$.

We next backtrack to decision level 3, unsetting e_7, e_9, e_{10} and e_{12}. Unit propagation at decision level 3 sets the new literal x and the first UIP $\overline{e_7}$ false and then sets literals e_9, e_{10}, and $\overline{e_{12}}$ true. This yields a contradiction with the clause $\overline{e_{10}} \vee e_{12}$. In the conflict graph at decision level 3, the first UIP is e_6 and there are two DIPs available, namely $\{\overline{e_7}, \overline{e_{12}}\}$ and $\{e_{10}, \overline{e_{12}}\}$. If we select the first one, then we introduce a new literal y defined by extension as $y \leftrightarrow \overline{e_7} \wedge \overline{e_{12}}$ and can in addition learn the clauses $e_3 \vee e_4 \vee \overline{e_5} \vee \overline{e_6} \vee y$ and $\overline{e_5} \vee \overline{y}$.

We do not carry this example further, but note that our experiments show that Tseitin tautologies (not just on grid graphs) are examples where our experiments show the DIP clause learning method is especially effective.

It is interesting to relate the Figure 6(b) to Theorem 4.4 on DIPs. In this example, there is only one way to choose the two vertex-disjoint paths. Namely, to let π_a and π_b be the paths $e_6, e_8, e_{11}, \overline{x}, \overline{e_7}, e_{10}, \bot$ and $e_6, e_9, \overline{e_{12}}, \bot$. The edge from e_6 to $\overline{e_7}$ bypasses e_8, e_{11}

and $\overline{x}$; and the edge from e_{11} to $\overline{e_{12}}$ is a crossing separator. (The edge from $\overline{e_7}$ to e_9 is a "vacuous" crossing separator that does not actually remove any possible DIP pairs.)

It should be evident from this example that many conflicts have DIPs; indeed our experiments reported below show that approximately 2/3 of the conflicts have at least one DIP and, very frequently there are quite a few choices for DIPs.

5.2. **Extending CDCL with Dual Implication Points.** The use of DIPs in conflict analysis opens a large spectrum of possibilities. This section discusses some of them, with particular attention to the techniques that we have implemented. Our present implementation, xMapleLCM, is a flexible framework that allows one to implement extended-resolution based techniques in a simple way. It offers a set of clearly-specified functions that facilitate determining which extension variables to add, performing the corresponding addition, the possible posterior deletion or replacing definitions in clauses. Additionally, a set of heuristic choices to control when and how these steps are performed are provided, and replacing them by custom ones is a smooth task. We want to remark that there are many more possible strategies for using DIPs than could be discussed. We believe that in this paper we merely scratched the surface of heuristics for exploiting DIPs, which is an indication of the potential of this approach.

Choice of DIP. As mentioned, there is possibly a quadratic number of DIPs. Even though we could learn multiple DIPs at every conflict, with their corresponding lemmas, we decided to choose only one. The first possibility we considered is to learn the DIP that is closest to the conflict, as we do with UIPs. This may often create a short post-DIP conflict but a long pre-DIP clause. Therefore, we considered the possibility of choosing a DIP that splits the conflict graph into two balanced regions. Ideally, that would result in two equally long pre and post-DIP clauses. Finally, we also implemented choosing a random DIP, to check whether any of the other two schemes could outperform a random strategy. These heuristics are referred to as **closest**, **middle**, and **random**, respectively.

Filtering out bad-quality DIPs. Learning a DIP whenever we find one would be too prolific and overwhelm the solver. In order to determine whether the DIP chosen in the previous step had to be discarded, the first possibility we explored was again inspired by 1-UIP learning, where learning glue clauses, i.e. having LBD equal to 2, is a desired situation. The first filtering mechanism we implemented discarded all DIPs that did not have a glue post-DIP clause. Another possibility we considered is to wait for a DIP to occur a certain number of times before using it in DIP-based learning. In our implementation, we tried different numbers as the minimum threshold to introduce a DIP. A third possibility is to use the activity-based heuristic of the literals in the DIP to assess its quality: DIPs whose literals have high decision-heuristic scores should be prioritized. In our implementation, we check whether the activity level of the current DIP is higher than the average activity level of the 20 most recently encountered DIPs; if so, the current DIP is a candidate for DIP-learning, otherwise it is discarded. We refer to these various techniques as **glue**, **occ** and **act**, respectively.

Learning pre-DIP and post-DIP clauses. In our implementation, we only considered two variants: one that always learns both the pre- and the post-DIP clauses (**2-clause**) and one that only learns the post-DIP clause (**1-clause**).

Backjumping and asserting clauses. Recall that when a new DIP extension variable z is introduced, it is possible to learn a pre-DIP clause of the form $\neg f \vee \neg C \vee z$ and a post-DIP clause of the form $\neg z \vee \neg D$, where f is the first UIP and where C and D are conjunctions of literals that were set true at lower levels. (Note that C and D may have literals in common.) Letting ℓ_C and ℓ_D be the maximum of the levels at which literals in C and D (respectively) were set, our implementation always backtracks to level ℓ_D. This makes the post-DIP clause asserting, so $\neg z$ is set at decision level ℓ_D. Furthermore, if $\ell_C \leq \ell_D$ and the pre-DIP clause is learned, then it is asserting and $\neg f$ is set by unit propagation at decision level ℓ_D, as it should be.

It would make no sense to backtrack to level ℓ_C when $\ell_C < \ell_D$ since then neither $\neg z$ nor $\neg f$ would be propagated. However, another possible strategy would be to backtrack to the maximum of the decision levels ℓ_C and ℓ_D. This would mean $\neg z$ and $\neg f$ are both propagated. The disadvantage of this when $\ell_C > \ell_D$ is that it would mean $\neg z$ is asserted by the post-DIP clause at level ℓ_C, whereas it could have been propagated at the previous decision level ℓ_D. This breaks a usual invariant for CDCL solvers. This would only be possible in a solver that permits chronological backtracking [NR18, MB19]; xMapleLCM, however, does not support this.

The previous reasoning needs some clarification for the case when the extension variable z with definition $z \leftrightarrow l_1 \wedge l_2$ we want to use has already been introduced. If z is undefined or defined at the current decision level nothing changes. We know it cannot be true at some previous level, because otherwise l_1 and l_2 would have been propagated at that level, and not in the current one as all literals belonging to a DIP do. If it is false at some previous level, then we have no guarantee that the pre or the post-DIP clause is asserting at any decision level. Fortunately, that rarely happens in practice. However, we can always perform standard 1UIP learning or try to apply DIP-based conflict analysis starting with the clause $z \vee \neg l_1 \vee \neg l_2$ that we can guarantee is conflicting at the current decision level.

Replacing literals by extended variables. In xMapleLCM, every time a new lemma is learned, we try to replace some of its literals by extended variables. An extended variable $z \leftrightarrow l_1 \vee l_2$ allows one to replace a lemma of the form $l_1 \vee l_2 \vee C$ by $z \vee C$. This is done by checking, for all pairs of literals in the lemma that appear in some extended variable definition whether they are part of the same definition. Lemmas that are too long or have large LBD are discarded to mitigate the cost of this operation.

If one wants to introduce literal $\neg z$ and obtain some reduction in formula size, the lemma should be of the form $\neg l_1 \vee C$ and there should be another clause of the form $\neg l_2 \vee C$. In this case, they can be replaced by a single clause $\neg z \vee C$. However, this situation can be expensive to detect since one has to traverse the whole database looking for a certain clause, and this operation should be repeated for every literal in the lemma. For this reason, xMapleLCM does not implement this.

Deleting extended variables. As it happens with lemma learning, where keeping too many lemmas slows down unit propagation, managing too many extended variables might also be counterproductive. Again following the analogy with learned lemmas, which are useful at some point of the search but might become inactive after a while, it is natural to think that extended variables follow the same behavior. All in all, it seems mandatory to consider the deletion of extended variables, which amounts to deletion all lemmas where they appear.

Deletion of variables in xMapleLCM is scheduled to be performed every 1000 conflicts. At that point, several strategies are possible: delete all variables, delete the ones with a minimum decision-heuristic activity, or delete the worst $k\%$ variables according to some criterion (e.g. their decision-heuristic activity). In our implementation, we do the latter with $k = 50$. Note that variables appearing in the right-hand side of an extended-variable definition cannot be deleted. This is addressed by maintaining a counter for every variable that corresponds to the number of definitions where it is involved.

6. Experimental Evaluation

We have implemented the DIP-based clause learning schemes described in Section 5.2 on top of the xMapleLCM ER framework[4]. We started our experimental evaluation running a variant of DIP-based learning on benchmarks of the SAT Competition [BHI+23] and comparing it to MapleLCM [LLX+17], the CDCL SAT solver on which it is based. Even though there did not seem to be a systematic improvement on all benchmarks, a few families with important speedups were identified. We start this section by analyzing the impact of DIP-based learning on these families and then move to final considerations about the performance on the overall SAT competition benchmarks.

For each benchmark family, we start by describing the problem they encode. After that, we evaluate the impact of the different techniques explained in Section 5.2. In particular, we first consider as a **baseline** a version that (i) finds DIPs in the **middle** of the conflict graph, (ii) only adds a DIP if it has occurred at least 20 times and (iii) always learns both the pre- and post-DIP learning clauses. Different variants are obtained by changing only one of the three previous design decisions at a time. Regarding the type of DIP used, we analyze the performance of **closest**, **random**, and **heuristic** , that is, the systems whose only difference w.r.t. **baseline** is that the type of DIP is changed. Regarding the criterion used to discard a DIP, we implemented the **glue** and the **act** configurations, but due to their poor performance we do not include these results. We present instead results about for the **occ5** and **occ50** configurations, which discard any DIP that has not occurred at least 5 or 50 times, respectively. Finally, we evaluated the performance of the **1-clause** variant only learns the post-DIP clause.

After that, we report on the performance of a variety of state-of-the-art solvers, each with some distinguished characteristic: Kissat 4.0.2 [BFFH20] (an extremely efficient CDCL solver), CryptoMiniSat 5.12.1 [SNC09] (support for XOR reasoning), SBVA-CaDiCaL [HGH23] (introduction of new variables via SBVA and winner of the main track of the 2023 SAT Competition), GlucosER [AKS10] (extended-resolution based CDCL solver) and the best available configuration for xMapleLCM.

6.1. Tseitin Formulas.

These formulas have already been described in Section 5.1. In this section, we consider three types of unsatisfiable formulas generated by CNFgen [LENV17]: (i) Tseitin formulas on the grid, where the graph is a rectangular grid and each vertex is connected to its 4 neighbors, except for the vertices on the boundaries of the grid which have fewer neighbors, (ii) 4-regular Tseitin formulas, where the graph is a 4-regular random one, and (iii) 6-regular Tseitin formulas built on top of 6-regular random graphs. The time limit was set to 600 seconds per instance.

The first row of Figure 7 shows the effect of changing the type of DIP that is used on our **baseline** solver. A point (x, y) indicates that this particular variant took y seconds (note the logarithmic scale on the y axis) to solve the corresponding Tseitin formula of size

[4]All sources used for this evaluation can be found in `https://github.com/chjon/xMapleSAT/tree/main`.

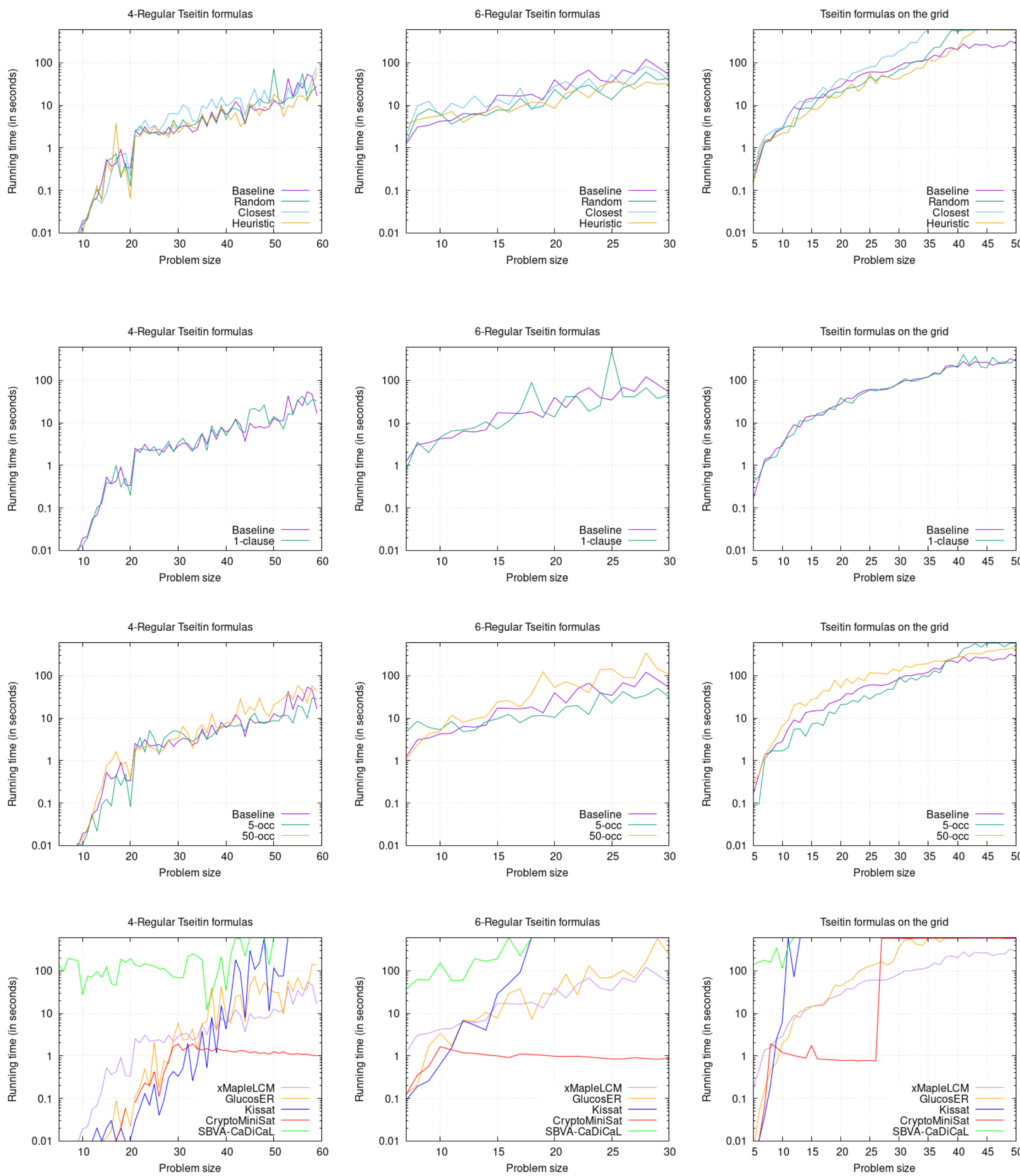


Figure 7: Performance of different xMapleLCM configurations and state-of-the-art solvers on Tseitin formulas.

x. We observe that there is no big difference for regular Tseitin formulas. However, for formulas on the grid, the **middle** DIP is the one that scales best, being the only variant able to solve all instances within the time limit. In the second row of the same figure, we can see how learning both the pre- and the post-DIP clauses (**baseline** in the plots) is comparable to only learning post-DIP clause (**1-clause**). Finally, the third row shows that changing the minimum number of times a DIP has to occur before we introduce it has some impact. 50

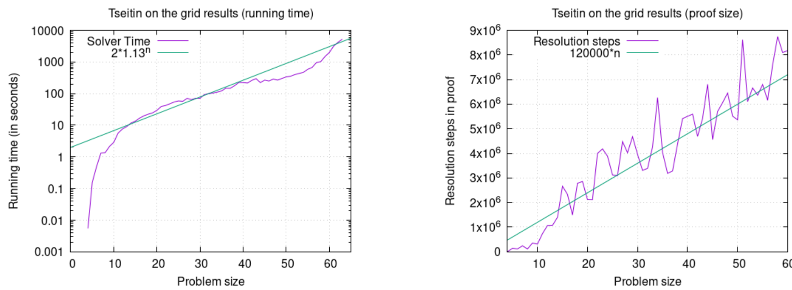


Figure 8: Performance analysis of the baseline DIP-based solver on Tseitin formulas on the grid. The plot on the left represents running time (with logarithmic scale on y), whereas the one on the right shows the number of resolution steps in the generated proof.

seems to be too restrictive, whereas 20 (**baseline**) is slightly better than 5, which might end up adding too many DIPs.

The last row of Figure 7 compares the **baseline** configuration of xMapleLCM with state-of-the-art tools. The conclusion is clear: our DIP-based xMapleLCM solver is the only one that can process all instances within the time limit. GlucosER scales well on regular formulas, but not on formulas on the grid. Despite these formulas are in principle trivial for CryptoMiniSat, since the application of Gaussian elimination in a preprocessing step solves them, this does not seem to work for large formulas on the grid. Finally, Kissat and SBVA-CaDiCaL perform poorly, solving only very small instances.

In order to understand whether the behavior of xMapleLCM was polynomial for these formulas, we generated more challenging Tseitin formulas on the grid. Results can be seen in the left plot of Figure 8, with logarithmic scale on the y axis. There is little doubt that the runtime of our solver ends up being exponential w.r.t. the size of the problem. However, we went one step further and studied how large the generated trimmed DRAT proofs were. On the right plot, we show the number of resolution steps in the proof. Despite having no theoretical support for that, it seems that although the solver takes exponential time in finding a proof, its size might be polynomial w.r.t. the problem size. That would indicate that our problem is using search heuristics that are not good enough to quickly find a short proof.

6.2. **Xorified Random k-XOR Formulas.** Another family where DIP-based systems perform very well are random k-xor formulas, where xorification [BS09] has been applied, i.e., replacing variables by xors of fresh variables. We used CNFgen to obtain 2 sets of formulas: xor constraints of length 3 applying xorification with 2 and 3 variables. All formulas we consider are unsatisfiable, the number of variables before xorification is equal to the number of clauses, and we increase this number to get progressively more difficult formulas. A time limit of 600 seconds was used.

The performance of the different xMapleLCM configurations can be seen in Figure 9. In the first row, we notice that there is not much difference if we change the DIP choice. A similar phenomenon can be observed in the second row, where the use of one or two learned DIP clauses does not have big impact on the performance. In the third row, we can see that

	Baseline	**Random**	**Closest**	**Heuristic**
SOLVED	6	**13**	10	**13**
Average (secs.)	6011	4337	4818	**4220**

	Heuristic	**1-clause**	**Occ-5**	**Occ-50**
SOLVED	13	11	6	**14**
Average (secs.)	4220	4995	6057	**3913**

	Occ-50	GlucosER	Kissat	CryptoMiniSat	SBVA-CaDiCaL
SOLVED	**14**	**14**	0	0	0
Average (secs.)	3913	**3622**	7200	7200	7200

Table 1: Average running times in seconds of DIP-learning variants on properly intersecting intervals formulas. The best performing systems are in bold.

changing the minimum number of occurrences of DIPs does affect the speed of the system: choosing 20 seems to give more consistent behavior.

Finally, in the last row of Figure 9 we notice that CryptoMiniSat can only benefit from its preprocessing step when xorification with 2 variables is applied. Our xMapleLCM solver is the one the performs best with GlucosER in second place. Finally, both SBVA-CaDiCaL and Kissat are only able to solve small instances.

6.3. **Matching of Properly Intersecting Intervals.** We are given a sequence of numbers $(a_1, \ldots, a_n)$, initially all set to zero, and a set of operations, each consisting of assigning a certain number to a contiguous subsequence of the a_i's, defined by an interval. The goal is to perform each operation exactly once while maximizing the number of pairs of consecutive numbers (a_i, a_{i+1}) that are different at the end of the process. Given some additional conditions, this can be formulated as maximum bipartite matching problem on a certain graph. By removing some edges from this graph, an unsatisfiable problem is generated. A more detailed description can be found in [BHI$^+$23].

We downloaded the 23 instances submitted to the SAT Competition, using the Global Benchmark Database [IS19], and observed that no system without extended-resolution reasoning could solve any benchmark in a time limit of 2 hours. We used a much larger timeout than with the two previous families because for this family we are not able to construct increasingly larger benchmarks and see how the solvers scale.

Table 1 reports on our experimental evaluation. In the first block we can see that the **random** and **heuristic** DIP choices do significantly improve upon our **baseline** solver. The second block row shows how the behavior of the **heuristic** configuration can be further improved by increasing the threshold of minimum DIP occurrences to 50. On the contrary, a smaller threshold of 5 significantly degrades the performance, whereas only learning the post-DIP clause slightly slows it down. Finally, the third block compares our best xMapleLCM configuration with state-of-the-art solvers. xMapleLCM and GlucosER are the only systems able to process any instance within the given time limit. Hence, the use of extended resolution is crucial in this set of benchmarks. Our comparison shows that GlucosER and xMapleLCM solve the same number of instances, the former being only slightly faster.

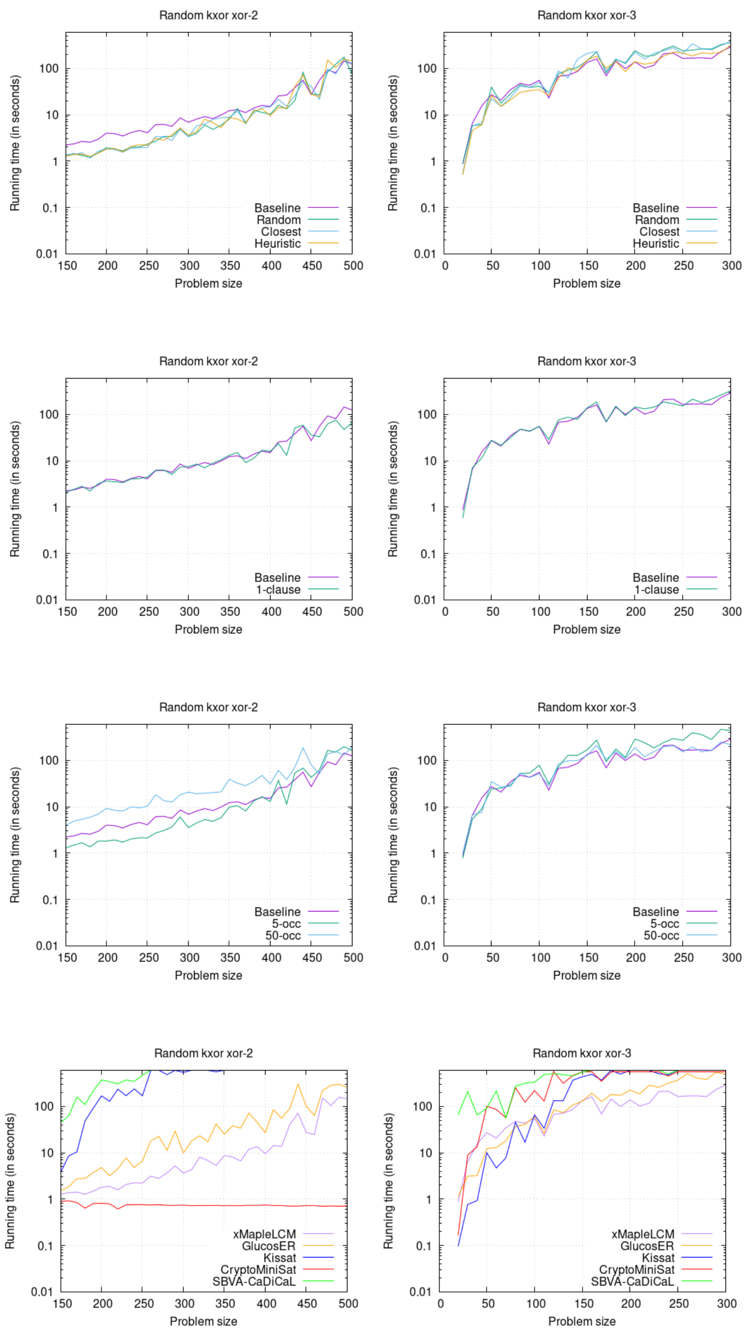


Figure 9: Performance of different xMapleLCM configurations and state-of-the-arts solvers on XORified rand-kxor formulas.

6.4. **Performance Analysis on SAT Competition Benchmarks.** Our experimental evaluation concludes with the lessons we have learned from executing our DIP implementation on the SAT competition benchmarks. We used benchmarks from the 2023 and 2024 editions. In the latter, the number of instances from the Tseitin, Xorified random k-xor and Interval families we have described is much lower than in the 2023 edition, and hence we present results from the 2023 competition in order to show the possible gains or losses of adding DIP-based reasoning to solvers. We first report on the overhead caused by the DIP detection algorithm and the subsequent additional work to retrieve the clauses to be learned. For our baseline DIP-based system, where two clauses are learned and hence is the most computationally demanding method, in 4.5% of the benchmarks the DIP-related work represented between 10 and 15% of the total runtime; in 11.5% of the benchmarks between 5 and 10%; in 35.5% between 2.5 and in 48.5% less than 2.5%. These data show that DIP computation does not significantly slow down the solver. Another interesting information concerns the percentage of conflicts where there is at least one DIP, which was on average 63%. This implies that we do need to have filtering mechanisms to discard some of them. Otherwise, the search would be totally dominated by DIPs.

In the left CDF plot of Figure 10 we can see that our baseline DIP-based xMapleLCM solver outperforms the CDCL-based MapleLCM that it is based on. When we analyzed the concrete benchmarks where our DIP-based solver outperforms MapleLCM and other CDCL-based solvers like Kissat, we observed that for all of them the percentage of decisions on extended variables by xMapleLCM is quite large (over 10% of the decisions), whereas for the remaining benchmarks it is very low (in 70% of the benchmarks less than 1% of decisions are on extended variables). Moreover, this happens no matter how many initial variables there are, even though in problems with few initial variables the introduction of a few extended variables could potentially quickly dominate the decision heuristic.

This is important since it shows that already existing decision heuristics like VSIDS or LRB somehow infer whether the newly introduced variables improve the behavior of the system. This is why we implemented, on top of our baseline DIP system, a procedure that computes the percentage of decisions on extended variables. If after a fixed number of conflicts it is still lower than 3%, it discontinues DIP learning and performs 1-UIP learning from that moment onwards. This is a very preliminary step in the direction of trying to automate the decision of whether to apply DIP reasoning or not, but the outcome, which is found on the left CDF plot of Figure 10, is very positive, showing that a solver disabling DIP reasoning (xMapleLCM-disabling in the plot) when the percentage of decisions on extended variables is low is still able to solve Tseitin, random k-xor and Interval benchmarks, but the overall performance is not significantly worsened on the remaining instances. Finally, the right CDF plot shows the impact of changing the type of DIP into the disabling solver: the **heuristic** and **middle** DIPs, the latter being used in the baseline configuration, are much better than **random** and **closest** ones.

Learned lessons from experimental evaluation. Let us finish this section by pointing out the main lessons we have learned from this experimental evaluation.

- We have identified three families of benchmarks for which the dynamic addition of extension variables improves upon CDCL in a systematic manner. This is an improvement w.r.t. the results in the GlucosER paper [AKS10], where a few initial benchmarks were identified but it was not proved that extended-resolution methods always outperform CDCL on increasingly more complex instances.
- We have proved that, even though the way extended variables are identified is completely different in xMapleLCM and GlucosER, they both behave well on the same instances. On the other hand, our preliminary experiments revealed that randomly choosing arbitrary pairs of (non-DIP) literals to create definitions does not provide any speedup w.r.t. CDCL.

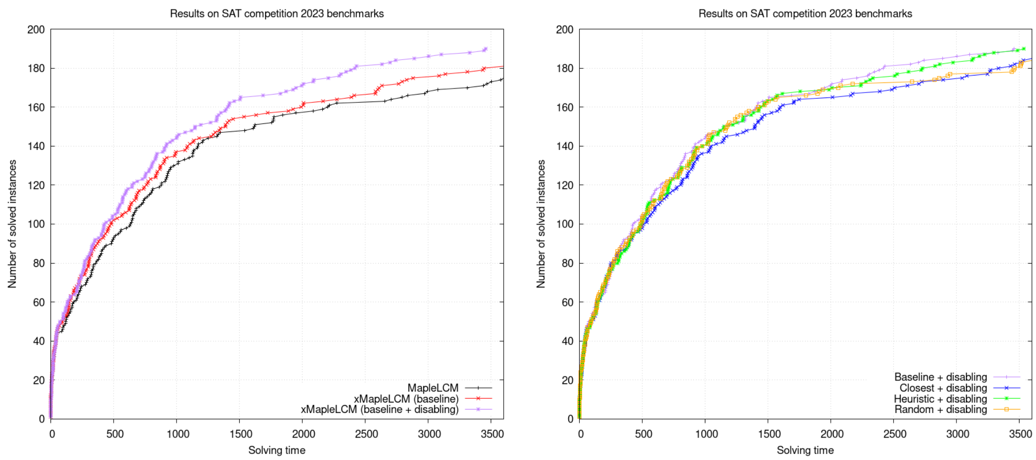


Figure 10: Performance of DIP variants on 2023 SAT Competition benchmarks.

Hence, both GlucosER and xMapleLCM are able to identify meaningful semantic properties about concrete literals that make them suitable for creating an extended variable definition. The data we collected show that GlucosER typically introduces around four times more extension variables than xMapleLCM and that the extended variables generated by GlucosER and xMapleLCM are neither disjoint nor totally overlap. Only about 25% of the definitions over original variables introduced in xMapleLCM are also introduced in GlucosER.

- We have learned that in most of the families where extended resolution works there is a large xor component. Additionally, when this is xor information is somehow hidden in the formula, our dynamic approach outperforms static methods like the ones in CryptoMiniSat and SBVA-CaDiCaL.
- Our ablation experiments have shown that the most important design choice is the one of determining which DIP is to be chosen. Among all the possibilities we have devised, **middle** and **heuristic** are the preferable ones.
- Finally, we have noticed that by inspecting the percentage of decisions on extended variables, one can detect whether our DIP-based is helping the solver. This has allowed us to further improve the behavior of our solver on sets of benchmarks where only a few of them are advantageous for extended resolution.

7. Conclusions and Future Work

We introduce a novel extended resolution clause learning (ERCL) algorithm that, when implemented on top of a CDCL solver, turns out to be beneficial for a variety of problems, in particular Tseitin, random k-xor and interval matching formulas. We view this as a step towards more effective methods for incorporating extended resolution to form more powerful proof systems. We show that the only previously existing attempt to incorporate ER into CDCL performs reasonably well on the same instances. Hence, we have identified classes of formulas for which extended resolution can be automated in at least two different ways and still outperform CDCL. Considering the different nature of the two methods, this deserves further study. Further, we also introduce a new heuristic that allows our ERCL solver xMapleSAT to perform similarly to the baseline CDCL solver MapleLCM, thus being able to get the best of both worlds, i.e., the benefit of ERCL, without sacrificing performance of the CDCL solver on, say, SAT Competition Main Track 2023 instances.

As future work, we plan to investigate a variety of machine learning based heuristics (e.g., branching) specialized for the ERCL method. Also, the use of k-IPs with $k > 2$ is part of our next steps. Finally, we intend to evaluate the impact of our technique on benchmarks from approximate hash-based counting, for which XOR reasoning is crucial. On the theoretical side, challenging questions like determining whether DIP- or k-IP based ER simulates unrestricted ER are going to be central to our research efforts.

Acknowledgments. We thank the anonymous referees for helpful comments and corrections.

References

[AFT11a] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Intelligence Research*, 40:353–373, January 2011. Preliminary version in *SAT '09*.

[AFT11b] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artif. Intell. Res.*, 40:353–373, 2011. doi:10.1613/jair.3152.

[AKS10] Gilles Audemard, George Katsirelos, and Laurent Simon. A restriction of extended resolution for clause learning sat solvers. In *AAAI*, 2010.

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009. URL: http://ijcai.org/Proceedings/09/Papers/074.pdf.

[Bac02] Fahiem Bacchus. Enhancing Davis Putnam with extended binary clause learning. In *Proc. 18th Annual Conference on Artificial Intelligence and 15th Conference on Innovative Applications of Artificial Intelligence*, pages 613–619. AAAI Press and MIT Press, 2002.

[BFFH20] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[BFW23] Armin Biere, Nils Froleyks, and Wenzi Wang. CadiBack: Extracting backbones with CaDiCal. In *Proc. 26th Intl. Conf. on Theory and Applications of Satisfiability Testing SAT*, pages 3:1–12. Leibniz-Zentrum für Informatik, 2023.

[BGO24] Sam Buss, Vijay Ganesh, and Albert Oliveras. A linear time algorithm for two-vertex bottlenecks. In preparation. Preliminary version available at https://math.ucsd.edu/~sbuss/ResearchWeb/TwoVertBottlenecks, March 2024.

[BH23] Randal E. Bryant and Marijn J. H. Heule. Generating extended resolution proofs with a bdd-based SAT solver. *ACM Trans. Comput. Log.*, 24(4):31:1–31:28, 2023. doi:10.1145/3595295.

[BHI+23] Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Finland, 2023.

[BKW24] Sam Buss, Oliver Kullmann, and Virginia Vassilevska Williams. Dual depth first search for binary clause reasoning. Preliminary version, March 2024.

[BS09] Eli Ben-Sasson. Size space tradeoffs for resolution. *SIAM Journal on Computing*, 38(6):2511–2525, 2009.

[Coo76] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, 1976. doi:10.1145/1008335.1008338.

[Hak85] Armin Haken. The intractability of resolution. *Theoretical computer science*, 39:297–308, 1985.

[HGH23] Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. Effective auxiliary variables via structured reencoding. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 11:1–11:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: `https://doi.org/10.4230/LIPIcs.SAT.2023.11`, `doi:10.4230/LIPICS.SAT.2023.11`.

[HKB17] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, 2017. `doi:10.1007/978-3-319-63046-5\_9`.

[HKB19] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Encoding redundancy for satisfaction-driven clause learning. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference (TACAS 2019), Part I*, Lecture Notes in Computer Science 11427, pages 41–58. Springer Verlag, 2019.

[HKB20] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *J. Autom. Reason.*, 64(3):533–554, 2020. `doi:10.1007/s10817-019-09516-0`.

[HKSB17] Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. Pruning through satisfaction. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, volume 10629 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2017. `doi:10.1007/978-3-319-70389-3\_12`.

[Hua10] Jinbo Huang. Extended clause learning. *Artificial Intelligence*, 174:1277–1284, 2010.

[IS19] Markus Iser and Carsten Sinz. A problem meta-data library for research in sat. In Daniel Le Berre and Matti J\"arvisalo, editors, *Proceedings of Pragmatics of SAT 2015 and 2018*, volume 59 of *EPiC Series in Computing*, pages 144–152. EasyChair, 2019. URL: `https://easychair.org/publications/paper/jQXv`, `doi:10.29007/gdbb`.

[Kau06] Henry A. Kautz. Deconstructing planning as satisfiability. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 1524–1526. AAAI Press, 2006. URL: `http://www.aaai.org/Library/AAAI/2006/aaai06-241.php`.

[Kro21] Daniel Kroening. Software verification. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 791–818. IOS Press, 2021. `doi:10.3233/FAIA201004`.

[KS92] Henry A. Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings*, pages 359–363. John Wiley and Sons, 1992.

[LENV17] Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. Cnfgen: A generator of crafted benchmarks. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 464–473. Springer, 2017. `doi:10.1007/978-3-319-66263-3\_30`.

[LLX+17] Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 703–711. ijcai.org, 2017. URL: `https://doi.org/10.24963/ijcai.2017/98`, `doi:10.24963/IJCAI.2017/98`.

[MB19] Sibyle Möhle and Armin Biere. Backing backtracking. In *22nd Intl Conf. on Theory and Applications of Satisfiability Testing (SAT)*, Lecture Notes in Computer Science 11628, pages 250–266. Springer, 2019.

[Men27] Karl Menger. Zur algemeinen Kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927.

[MHB12] Norbert Manthey, Marijn Heule, and Armin Biere. Automated reencoding of boolean formulas. In Armin Biere, Amir Nahir, and Tanja E. J. Vos, editors, *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November*

6-8, 2012. Revised Selected Papers, volume 7857 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2012. doi:10.1007/978-3-642-39611-3_14.

[MLM21] João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.

[NR18] Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In *21nd Intl Conf. on Theory and Applications of Satisfiability Testing (SAT)*, Lecture Notes in Computer Science 10929, pages 111–121. Springer, 2018.

[OLW+23] Albert Oliveras, Chunxiao Li, Darryl Wu, Jonathan Chung, and Vijay Ganesh. Learning shorter redundant clauses in SDCL using maxsat. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 18:1–18:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.SAT.2023.18, doi:10.4230/LIPICS.SAT.2023.18.

[PD08] Knot Pipatsrisawat and Adnan Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1481–1484. AAAI Press, 2008. URL: http://www.aaai.org/Library/AAAI/2008/aaai08-243.php.

[PD10] Knot Pipatsrisawat and Adnan Darwiche. On modern clause-learning satisfiability solvers. *Journal of Automated Reasoning*, 44(3):277–301, 2010.

[PD11] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011. doi:10.1016/j.artint.2010.10.002.

[RHB22] Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. Preprocessing of propagation redundant clauses. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 106–124. Springer, 2022. doi:10.1007/978-3-031-10769-6_8.

[Sak21] Karem A. Sakallah. Symmetry and satisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 509–570. IOS Press, 2021. doi:10.3233/FAIA200996.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009. doi:10.1007/978-3-642-02777-2_24.

[Tse83] Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pages 466–483, 1983.

[Urq87] Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987. doi:10.1145/7531.8928.

[XA05] Yichen Xie and Alexander Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *Proceedings of the 17th International Conference on Computer Aided Verification, CAV 2005*, pages 139–143, 2005. doi:10.1007/11513988_13.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In Rolf Ernst, editor, *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001*, pages 279–285. IEEE Computer Society, 2001. doi:10.1109/ICCAD.2001.968634.