Clause size reduction with i-UIP Learning

Nick Feng and Fahiem Bacchus

Department of Computer Science, University of Toronto, Canada,
{nfeng,fbacchus}@cs.toronto.edu

Abstract.

1 Introduction

Clause learning is an essential technique in SAT solvers. There is good evidence to indicate that it is, in fact, the most important technique used in modern SAT solvers [4]. In early SAT research a number of different clause learning techniques were proposed [14,10,11,3]. However, following the revolutionary performance improvements achieved by the Chaff SAT solver, the field has converged on using the 1-UIP (first Unique Implication Point) scheme [14] employed in Chaff [7] (as well as other techniques pioneered in the Chaff solver).[1] Since then almost all SAT solvers have employed the 1-UIP clause learning scheme, along with clause minimization [12], as their primary method for learning new clauses.

However, other clause learning schemes can be used in SAT solvers without changes to their main data structures. Furthermore, advances in our understanding allow us to better understand the potential advantages and disadvantages of these alternate schemes. In this paper we reexamine some of these previous clause learning schemes, with a focus on the schemes described in [14]. Improved understanding of SAT solvers, obtained from the last decade of research, allows us to see that in their original form these other clause learning schemes suffer significant disadvantages over 1-UIP clause learning.

One of the previously proposed schemes was the i-UIP scheme [14]. In this paper we propose a new way to exploit the main ideas of this scheme that avoids its main disadvantages. In particular, we propose to use a i-UIP like clause learning scheme to generate smaller learnt clauses which retain the good properties of standard 1-UIP clauses. Our method is related to, but not the same as various clause mininimization methods that aim to remove literals from the 1-UIP clause, e.g., [12,6,13]. Our method is orthogonal to clause minimization. In particular, our approach can learn a clause that is completely different from the 1-UIP clause but which still serves all of the same purposes as the 1-UIP clause. Like the 1-UIP clause, various minimization techniques can be applied on top of our method to further reduce the size of the clauses we learn.

We present various versions of our method and show that these variants are often capable of learning shorter clauses that the 1-UIP scheme, and that this can lead to useful performance gains in state of the art SAT solvers.

[1] The idea of UIP clauses was first mentioned in [11], and 1-UIP clauses along with other UIP clauses were learnt and used in the earlier GRASP SAT solver.

2 Clause Learning Framework

We first provide some background and a framework for understanding clause learning as typically used in CDCL SAT solvers. A propositional formula F expressed in Conjunctive Normal Form (CNF), contains a set of variables V. A literal is a variable $v \in V$ or its negation $\neg v$. For a literal ℓ we let var(ℓ) denote its underlying variable. A CNF consists of a conjunction of clauses, each if which is a disjunction of literals. We often view a clause as being a set of literals and employ set notation, e.g., $\ell \in C$ and $C' \subset C$.

Two clauses c_1 and c_2 can be *resolved* when they contain conflicting literals $\ell \in c_1$ and $\neg\ell \in c_2$. Their resolvant $c_1 \bowtie c_2$ is the new clause $(c_1 \cup c_2) - \{\ell, \neg\ell\}$. The resolvant is tautological if c_1 and c_2 contain more than one pair of conflicting literals.

We assume the reader is familiar with the operations of CDCL SAT solvers, and the main data structures used in such solvers. A good source for this background is [9].

The Trail. CDCL SAT solvers maintain a **trail**, $\mathcal{T}$, which is a *non-contradictory, non-redundant sequence of literals* that have been assigned TRUE by the solver; i.e. $\ell \in \mathcal{T} \rightarrow \neg\ell \notin \mathcal{T}$, and $\mathcal{T}$ contains no duplicates. Newly assigned literals are added to the end of the trail, and on backtrack literals are removed from the end of the trail and unassigned. If literal ℓ is on the trail let $\iota(\ell)$ denote its index on the trail sequence ($\mathcal{T}[\iota(\ell)] = \ell$). For convenience, we also let $\iota(\ell) = \iota(\neg\ell) = \iota(\text{var}(\ell))$, where var($\ell$) is ℓ's variable, even though $\neg\ell$ nor var(ℓ) are actually on $\mathcal{T}$. If x and y are both on the trail and $\iota(x) < \iota(y)$ we say that x *appears before* y *on the trail*. In a slight abuse of notation we also use $\iota(v)$ on a variable v where $\iota(v) = \iota(\ell)$ and ℓ a literal of v on the trail.

Two types of true literals appear on the trail: decision literals that have been assumed to be true by the solver, and unit propagated literals that are forced to be true because they are the sole remining unfalsified literal of a clause. Each literal $\ell \in \mathcal{T}$ has a decision level $decLvl(\ell)$ which is equal to the number of decision literals appearing before ℓ on the trail plus one; hence, $decLvl(d) = 1$ for the first decision literal $d \in \mathcal{T}$. The set of literals on $\mathcal{T}$ that have the same decision level forms a contiguous subsequence of $\mathcal{T}$ that starts with a decision literal d_i and ends just before the next decision literal d_{i+1}. If $decLvl(d_i) = i$ we call this subsequence of $\mathcal{T}$ the *i-th decision level*.

Each literal $\ell \in \mathcal{T}$ also has a clausal reason $reason(\ell)$. If ℓ is a unit propgated literal, $reason(\ell)$ is a clause of the formula such that $\ell \in reason(\ell)$ and $\forall x \in reason(\ell).\, x \neq \ell \rightarrow (\neg x \in \mathcal{T} \wedge \iota(\neg x) < \iota(\ell)$. That is, $reason(\ell)$ is a clause that has become unit implying ℓ due to the literals on the trail above ℓ. If ℓ is a decision literal then $reason(\ell) = \varnothing$.

In most SAT solver implementations clause learning is initiated as soon as a clause is falsified by $\mathcal{T}$. In this paper we will be concerned with the subsequent clause learning process which uses $\mathcal{T}$ to derive a new clause that can be added to the formula. In some implementations the trail might be altered during clause learning. Here, however, we will assume that $\mathcal{T}$ remains intact during clause learning and is only changed after the new clause is derived. After the new clause is lerned the $\mathcal{T}$ will be changed by backtracking.

Say that $\mathcal{T}$ falsifies a clause c_I, and that the last decision literal d_k in $\mathcal{T}$ has decision level k. Consider $\mathcal{T}_{k-1}$ the prefix of $\mathcal{T}$ above the last decision level, i.e., the sequence of literals $\mathcal{T}[0]$—$\mathcal{T}[\iota(d_k) - 1]$. We will assume that $\mathcal{T}_{k-1}$ is **propagation complete**, although the full trail $\mathcal{T}$ might not be. This means that (a) no clause was falsified by

$\mathcal{T}_{k-1}$. And (b) if c_u is a clause containing the literal x and all literals in c_u except for x are falsified by $\mathcal{T}_{k-1}$, then $x \in \mathcal{T}_{k-1}$ and $decLvl(x) \leq \max\{decLvl(y) | y \in c_u \wedge y \neq x\}$. This means that if x appears in a clause made unit it must have been added to the trail, and added at the first decision level that this occurred. Note, however, x might appear in more than one clause. The less than equal condition above ensures that x must appear in $\mathcal{T}_{k-1}$ at the first decision level any of these clauses were made unit.

Any clause falsified by $\mathcal{T}$ is called a **conflict**. When a conflict is found, the final level of the trail, k, need not be propagation complete as the solver typically stops propagation as soon as it finds a conflict. This means that (a) other clauses might be falsified by $\mathcal{T}$ besides the conflict found, and (b) other literals might be unit implied by $\mathcal{T}$ but not added to $\mathcal{T}$.

Definition 1 (Trail Resolvant) *A trail resolvant is a clause arising from resolving a conflict against the reason clause of some literal $\ell \in \mathcal{T}$ where $reason(\ell) \neq \varnothing$. Every trail resolvant is also a conflict.*

The following things can be noted about trail resolvants: (1) trail resolvants are never tautological, as the polarity of all literals in $reason(\ell)$ other than ℓ must agree with the polarity of all literals in the conflict (they are all falsified by $\mathcal{T}$); (2) one polarity of the variable var(ℓ) resolved on must be a unit propagated literal whose negation appears in the conflict; and (3) any literal in the conflict that is unit progagated in $\mathcal{T}$ can be resolved upon (the corresponding variable must appear in different polarities in the conflict and in $\mathcal{T}$).

Definition 2 (Trail Resolution) *A trail resolution is a sequence of trail resolvants applied to an initial conflict c_I yielding a new conflict c_L. A trail resolution is **ordered** if the sequence of variables $v_1, \ldots, v_m$ resolved have strictly decreasing trail indicies: $\iota(v_i) > \iota(v_{i+1})$ $(1 \leq i < m)$. (Note that this implies that no variable is resolved on more than once).*

Ordered trail resolutions resolve unit propagated literals from the end of the trail to the beginning. Without lost of generality, we can require that all trail resolutions be ordered.

Observation 1 *If the unordered trail resolution U yields the conflict clause c_L from an initial conflict c_I, then there exists an ordered trail resolution O that yields a conflict clause c'_L such that $c'_L \subseteq c_L$.*

Proof. Let U be the sequence of clauses $c_I = c_0, c_1, \ldots, c_m = c_L$ obtained by resolving on the sequence of variables $v_1, \ldots, v_m$ whose corresponding literals on $\mathcal{T}$ are $l_1, \ldots, l_m$. Reordering these resolution steps so that the variables are resolved in order of decreasing trail index and removing duplicates yields an ordered trail resolution O with the desired properties. Since no reason clause contains literals with higher trail indicies, O must be a valid trail resolution if U was, and furthermore O yields the clause $c'_L = \bigcup_{i=1}^{m} reason(l_i) - \{l_1, \neg l_1, \ldots, l_m, \neg l_m\}$. Since U resolves on the same variables (in a different order) we must that that the clause it produces $c_L \subseteq c'_L$. But c_L might retain one or more of the literals $\neg l_1, \ldots, \neg l_m$, since resolving on a literal l_j with higher trail index might reintroduce a literal $\neg l_i$ with lower trail index that was previously resolved away by an earlier resolution. □

The relevance of trail resolutions is that all proposed clause learning schemes we are aware of use trail resolutions to produce learnt clauses. Furthermore, the commonly used technique for clause minimization [12] is also equivalent to a trail resolution that yields the minimized clause from the un-minimized clause. Interestingly, it is standard in SAT solver implementations to perform resolution going backwards along the trail. That is, these implementations are typically using ordered trail resolutions. The above observation shows that this in fact the right way to do this.

Ordered trail resolutions are a special case of *trivial resolutions* [2]. Trail resolutions are specific to the trail data structure typically used in SAT solvers. If $\mathcal{T}$ falsifies a clause at its last decision level, then its associated implication graph [11] contains a conflict node. Cuts in the implication graph that separate the conflict from the rest of the graph correspond to conflict clauses [2]. It is not difficult to see that the proof Proposition 4 of [2] applies also to trail resolutions. This means that *any conflict clause in the trail's implication graph can be derived using a trail resolution.*

2.1 Some alternate Clause Learning Schemes

A number of different clause learning schemes for generating a new learnt clause from the initial conflict have been presented in prior work, e.g., [14,10,11,3]. Figure 1 gives a specification of some of these methods: (a) the all-decision scheme which resolves away all implied literals leaving a learnt clause over only decision literals; (c) the 1-UIP scheme which resolves away literals from the deepest decision level leaving a learnt clause with a single literal at the deepest level; (d) the all-UIP scheme which resolves away literals from each decision level leaving a learnt clause with a single literal at each decision level; and (e) the i-UIP scheme which resolves away literals from the i deepest decision levels leaving a learnt clause with a single literal at its i deepest decision levels. It should be noted that when resolving away literals at decision level i new literals at decision levels less than i might be introduced into the clause. Hence, it is important in the i-UIP and all-UIP schemes to use ordered trail resolutions.

Both the all-decision and all-UIP schemes yield a clause with only one literal at each decision level, and the all-UIP clause will be no larger that the all-decision clause. Furthermore, it is known [11] that once we reduce the number of literals at a decision level d to one, we could continue performing resolutions and later achieve a different single literal at the level d. In particular, a decision level might contain more than one unique implication point. The algorithms given in Figure 1 stop at the first UIP of a level, except for the all-decision schemes with stops at the last UIP of each level.

2.2 Asserting Clauses and LBD—Reasons to prefer 1-UIP clauses

An **asserting clause** [8] is a conflict clause C_L that has exactly one literal ℓ at its deepest level, i.e., $\forall l \in C_L . decLvl(l) \leq decLvl(\ell) \wedge (decLvl(l) = \ell \rightarrow l = \ell)$. All of the clause learning schemes in Figure 1 produced learnt clauses that are asserting.

The main advantage of asserting clauses is that they are 1-Empowering [8], i.e., they allow unit propagation to derive a new forced literal. Hence, asserting clauses can be used to guide backtracking—the solver can backtrack from the current deepest level to the point the learnt clause first becomes unit and then use the learnt clause to add a new

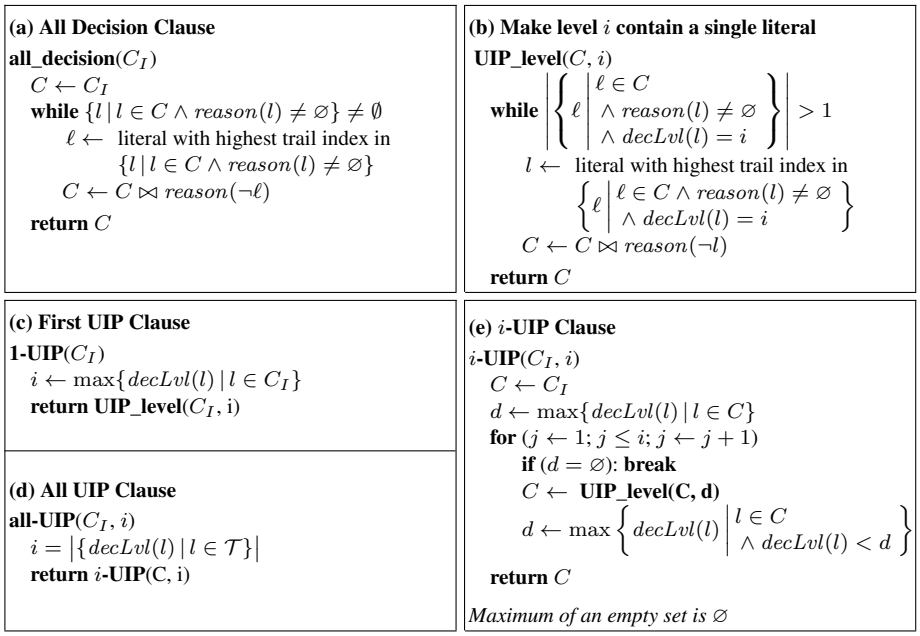

Fig. 1. Some different clause learning schemes. All use the current trail $\mathcal{T}$ and take as input an initial clause C_I falsified by $\mathcal{T}$ at its deepest level.

unit implicant to the trail. Since all but the deepest level was propagation complete, this means that the asserting clause must be a brand new clause; otherwise that unit implication would already have been made. On the other hand, if the learnt clause C_L is not asserting then *it could be that it is a duplicate of another clause already in the formula*. In particular, C_L contains at least two literals at the deepest level since it is not asserting. If two of its literals at the deepest level were were not completely unit propagated (unit propagation is aborted as soon as a conflict is found), then C_L could already be in the formula: it's two watches might not be fully propagated and C_L could be a falsified clause not detected by the solver. (In general, more than one clause might be falsified by $\mathcal{T}$ and the SAT solver will generally stop when it finds the first one).

The LBD of the learnt clause C_L is the number of different decision levels in it: $\text{LBD}(C_L) = \left|\left\{decLvl(l) \mid l \in C_L\right\}\right|$ [1]. Empirically LBD is a empirically successful predictor of clause usefulness: clauses with lower LBD tend to be more useful. As noted in [1] from the initial falsified clause C_I the 1-UIP scheme will produce a clause C_L whose LBD is minimum amongst all asserting clauses that can be learnt from C_I. If C' is a trail resolvant of C and a reason clause $reason(l)$ ($l \in C$), then $\text{LBD}(C') \geq \text{LBD}(C)$ since $reason(l)$ must contain at least one other literal with the same decision level as l. That is, each trail resolution step can only increase the LBD of the learnt clause. The 1-UIP scheme performs the minimum number of trail resolution steps required to generate an asserting clause.

Putting these two observations together we see that the 1-UIP scheme produces asserting clauses with lowest possible LBD. This is a compelling reasons for using this scheme. Hence, it is not surprising that modern SAT sovlers almost exclusively use 1-UIP clause learning.[2]

3 Using i-UIP Clause Learning

Although learning clauses with low LBD has been shown to be important in SAT solving [1], clause size is also important. Smaller clauses consume less memory and help to decrease the size of future learnt clauses. They are also semantically stronger than longer clauses.

The i-UIP scheme will tend to produce small clauses since the clauses can contain at most one literal per decision level. However, since more trail resolution steps are required to generate them they will also tend to contain more decision levels and thus have higher LBD. They will, however, be asserting.

Since, LBD has shown itself to be more important than size our approach is to use i-UIP learning when, and only when, it succeeds in reducing the size of the clause *without increasing its LBD*. It can be noted that the i-UIP scheme first computes the 1-UIP clause when it reduces the deepest level to a single UIP literal. It then proceeds to reducing the shallower levels (see i-UIP's for loop in Figure 1). So our approach will start with the normal 1-UIP clause and then try to apply i-UIP learning to reduce other levels to single literals. As noted above, clause minimization is orthogonal to our approach, so we also first apply standard clause minimization [12] to the 1-UIP clause. That is, our algorithm will start with the clause that most SAT solvers learn from a conflict (a minimized 1-UIP clause). Our approach is specified in Alg. 1.

The Algorithm **LBDstable-iUIP** attempts to reduce each level in the 1-UIP clause to a single literal (a UIP). The 1-UIP clause has only one literal at its deepest level, so that level, $decLvls[0]$, can be skipped. The subroutine **try-uip-level** is used to reduce each level. It uses trail resoluions to achieve this, subject to the constraint that no new decision levels can be introduced in the clause. In particular, **try-uip-level** (C_i, i) attempts to resolve away the literals at decision level i in the clause C_i, i.e., those in the set L_i (line 12), in order of decreasing trail index, until only one literal at level i remains. If the resolution step will not introduce any new decision levels (line 20), it is performed updating C_{try}. In addition, all new literals at level i are added to L_i. These can only be lits at level i in the reason clause.

On the other hand, if the resolution step would introduce new decision levels (line 15) then there are two options. The first option we call **Pure-iUIP**. With **Pure-iUIP** we abort our attempt to UIP this level and return the clause with level i unchanged. In the second option, called **Min-iUIP**, we continue without performing the resolution, *keeping* the current literal p in C_{try}. **Min-iUIP** will continue to try to resolve away the other literals in L_i (note that p is no longer in L_i) until L_i is reduced to a single literal. Hence, **Min-iUIP** can leave multiple literals at level i—all of those with reasons

[2] Knuth in his sat13 CDCL solver [5] uses an all-decision clause when the 1-UIP clause is too large. In this context an all-UIP clause could also be used as it would be no larger than the all decision clause.

Algorithm 1 LBDstable-iUIP

Require: C_1 is minimized 1-UIP clause

1: **LBDSTABLE-iUIP**($C_1, \mathcal{T}$, *type*)
2: $C_i \leftarrow C_1$
3: $decLvls \leftarrow$ decision levels in C_i in descending order ▷ These never change
4: **for** ($i = 1; i < |decLvls|; i$++) ▷ skip the deepest level $decLvls[0]$
5: $C_i \leftarrow$ **try-uip-level** (C_i, $decLvls[i]$, *type*) ▷ Try to reduce this level to UIP
6: **if** $\left|\{\ell \,|\, \ell \in C_i \wedge decLvl(\ell) \leq i\}\right| + (|decLvls| - (i+1)) \geq |C_1|$
7: **return** C_1. ▷ can't generate smaller clause
8: **if** (*type* = **Pure-iUIP**) $C_i \leftarrow \mathbf{miminize}(C_i)$
9: **return** if $|C_i| < |C_1|$) then C_i else C_1

10: **TRY-UIP-LEVEL**(C_i, i, *type*) ▷ Do not add new decision levels
11: $C_{try} = C_i$
12: $L_i = \{\ell \,|\, \ell \in C_{try} \wedge decLvl(l) = i\,\}$
13: **while** $|L_i| > 1$
14: $p \leftarrow$ **remove** lit with the highest trail index from L_i
15: **if** ($\exists q \in reason(\neg p).\ decLvl(q) \notin decLvls$) ▷ Would add new decision levels
16: **if** (*type* = **Pure-iUIP**)
17: **return** C_i ▷ Abort, can't UIP this level
18: **else if** (*type* = **Min-iUIP**)
19: **continue** ▷ Don't try to resolve away p
20: **else**
21: $C_{try} \leftarrow C_{try} \bowtie reason(\neg p)$
22: $L_i = L_i \cup \{\ell \,|\, \ell \in reason(\neg p) \wedge \ell \neq \neg p \wedge decLvl(\ell) = i\}$
23: **return** C_{try}

containing new levels along with one other. Note that since the sole remaining literal $u \in L_i$ is at a lower trail index than all of the kept literals, so there is no point in trying to resolve away u—either it will be the decision clause for level i having no reason, or its reason will contain at least one other literal at level i.

After trying to UIP each level the clause C_i is obtained. If we were using **Pure-iUIP** we can once again apply minimization (line 8). Recursive clause minimization [12] would be useless for the **Min-iUIP** clause as all but one literal of each level introduces a new level and thus cannot be recursively removed.[3]

Finally, the Algorithm returns the shorter of the initial 1-UIP clause and the newly computed clause (line 9). Because we return the newly computed clause only when it is shorter an early termination test can be used (line 7). After the algorithm has finished processing levels $decLvls[0]$–$decLvls[i]$ the literals at those levels cannot be further changed. Furthermore, we know that the best that can be done is to reduce the remaining $|decLvls| - (i+1)$ levels down to a single literal each. These two observations give a lower bound on the size of the resulting clause, and if that lower bound is as large as the size of the initial 1-UIP clause we can terminate and return the initial 1-UIP clause.

[3] Other more powerful minimization techniques could still be applied.

Example 1. Consider the trail $\mathcal{T} = \ldots, \ell_1, a_2, b_2, c_2, d_2, \ldots, \ldots, e_5, f_5, g_5, h_6, i_6, j_6, k_6, \ldots, m_{10}, \ldots$ where the subscript indicates the decision level of each literal and the literals are in order of increasing trail index. Let the clauses C_x denote the reason clause for literal x_i with

$C_a = \varnothing$	$C_b = (b_2, \neg\ell_3, \neg a_2)$	$C_c = (c_2, \neg a_2, \neg b_2)$
$C_d = (d_2, \neg b_2, \neg c_2)$	$C_\ell = \varnothing$	$C_e = \varnothing$
$C_f = (f_5, \neg e_5, \neg \ell_1)$	$C_g = (g_5, \neg a_2, \neg f_5)$	$C_h = \varnothing$
$C_i = (i_6, \neg e_5, \neg h_6)$	$C_j = (j_6, \neg f_5, \neg i_6)$	$C_k = (k_6, \neg f_5, \neg j_6)$

Suppose 1-UIP learning yields the clause $C_1 = (\neg m_{10}, \neg k_6, \neg j_6, \neg i_6, \neg h_6, \neg g_5, \neg d_2, \neg c_2)$ where $\neg m_{10}$ is the UIP from the conflicting level. **LBDstable-iUIP** first tries to find the UIP for level 6 by resolving C_1 with C_k, C_j and then C_i producing the clause $C^* = (\neg m_{10}, \neg h_6, \neg g_5, \neg f_5, \neg e_5, \neg d_2, \neg c_2)$ where $\neg h_6$ is the UIP for level 6.

LBDstable-iUIP then attempts to find the UIP for level 5 by resolving C^* with C_g and then C_f. However, resolving with C_f would introduce ℓ_1 and a new decision level into C^*. **Pure-iUIP** thus leaves level 5 unchanged . **Min-iUIP**, on the other hand, skips the resolution with C_f leaving f_5 in C^*. Besides f_5 only one other literal at level 5 remains in the clause, e_5, so **Min-iUIP** does not do any further resolutions at this level. Hence, **Pure-iUIP** yields C^* unchanged, while **Min-iUIP** yeilds $C^*_{min} = (\neg m_{10}, \neg h_6, \neg f_5, \neg e_5, \neg d_2, \neg c_2, \neg a_2)$.

Finally, **LBDstable-iUIP** processes level 2. Resolving away d_2 and then c_2 will lead to an attempt to resolve away b_2. But again this would introduce a new decision level with the literal ℓ_1. So **Pure-iUIP** will leave level 2 unchanged and **Min-iUIP** will leave b_2 unresolved. The final clauses produced by **Pure-iUIP** would be $(\neg m_{10}, \neg h_6, \neg f_5, \neg e_5, \neg d_2, \neg c_2, \neg a_2)$, a reduction of 1 over the 1-UIP clause, and by **Min-iUIP** would be $(\neg m_{10}, \neg h_6, \neg f_5, \neg e_5, \neg b_2, \neg a_2)$, a reduction of 2 over the 1-UIP clause. □

The core algorithm of **LBDstable-iUIP** as a clause reduction technique is simple. However, to make the algorithm a practical learning scheme that is compatible with modern SAT solvers, a number of issues need to be addressed. The rest of the section details the key optimizations and augmentations of **LBDstable-iUIP** as a practical clause learning scheme.

3.1 Control i-UIP Learning

This simple and greedy **LBDstable-iUIP** learning scheme can produce significantly smaller clause. However, when **LBDstable-iUIP** does not reduce the clause size, the cost of the additional resolution steps will hurt the solver's performance. Since resolution cannot reduce a clause's LBD, The maximum size reduction from **LBDstable-iUIP** is the difference between the clause's size and LBD, denote as the clause's gap value ($\text{Gap}(C_1) = |C_1| - \text{LBD}(C_1)$). For an 1-UIP clause with a small Gap, applying **LBDstable-iUIP** is unlikely to achieve cost effective results. Therefore, we propose a heuristic based approach to to enable and disable **LBDstable-iUIP** learning based on input clause's Gap.

Alg. 2 compares the input C_1's Gap against a floating target threshold t_{gap} (line 3). The threshold t_{gap} represent the expected minimal Gap required for C_1 to achieve a predetermined success rate (80%) from performing **LBDstable-iUIP** learning.

The gap threshold t_{gap} is initialized to 0, and is updated at every restart based on **LBDstable-iUIP**'s success rate from the previous restart interval. More specifically, the algorithm collects the statistics of the number of **LBDstable-iUIP** learning attempted (line 8 in alg. 2) and the number of attempts succeed (line 7) for each restart interval, and use them to calculate the success rate. If the success rate is below 80, the threshold t_{gap} is increased to restrict **LBDstable-iUIP**for the next restart interval. On the other hand, the threshold is decreased to encourage more aggressive **LBDstable-iUIP** learning for the next restart interval.

$$t_{gap} = \begin{cases} t_{gap} + 1 & \text{if } \frac{\text{Succeed}}{\text{Attempted}} < 0.8 \\ max(t_{gap} - 1, 0), & \text{otherwise} \end{cases}$$

3.2 Early stop i-UIP Learning

At any point of **Min-iUIP** learning, if the number of marked literals in C_i (literals which are forced into the clause to preserve LBD) exceeds the input clause C_1's Gap, we can abort **LBDstable-iUIP** learning for C_1 because the size of the final C_i is at least the size of C_1. The early stopping rule prevents solver from wasting time on traversing a large implication graph when **LBDstable-iUIP** has already failed.

The early stopping rule may prematurely stops **Pure-iUIP** learning which would otherwise succeed because C_i can be further minimized. However, the experiments indicates such false negative cases are rare, and the early stopping rule generally helps the learning scheme's performance.

3.3 Greedy Active Clause Selection

Even though **LBDstable-iUIP** learning can reduce the size of the learnt clause C_i, but it may introduce literals with low variable activity into C_i. Inactive literals prevents the clause from being asserted to force literal implication. Therefore, a practical clause

Algorithm 2 Control-**LBDstable-iUIP**

Require: C_1 is a valid 1-UIP clause
Require: $t_{gap} \geq 0$ is a dynamically calculated gap threshold
1: CONTROL-**LBDSTABLE-iUIP**(C_1, t_{gap})
2: $Gap \leftarrow |C_1| - LBD(C_1)$
3: **if** $Gap > t_{gap}$
4: $C_i \leftarrow$ **LBDstable-iUIP**(C_1, ϕ)
5: **I-UIP-Greedy**(C_i , C_1) ▷ additional clause selection policy, see sec 3.3
6: **if** $|C_i| < |C_1|$
7: Succeed $\leftarrow$ Succeed $+ 1$
8: Attempted $\leftarrow$ Attempted $+ 1$

Algorithm 3 i-UIP-Greedy

Require: C_i is a valid i-UIP clause
Require: C_1 is a valid 1-UIP clause
1: **i-UIP-GREEDY**(C_i, C_1)
2: **if** $|C_i| < |C_1| \wedge (\text{AvgVarAct}(C_i) > \text{AvgVarAct}(C_1))$
3: **return** C_i
4: **else**
5: **return** C_1

learning scheme should consider both size and variable activity. We propose an optional extension **i-UIP-Greedy** to filter out inactive C_1.

After computing the C_i, **i-UIP-Greedy** compares both the size and the average variable activity for C_1 and C_i (alg. 3 at line 2). The algorithm learns C_i if the clause has smaller size and higher average variable activity.

3.4 Adjust Variable Activity

Two popular branching heuristics for modern SAT solvers are VSIDS and LBR. Both heuristics increase the variable activity for all literals involved in resolutions during 1-UIP learning. Since **LBDstable-iUIP** extends 1-UIP with deeper resolutions against the trail, the variables activities for the fresh literals involved in the additional resolution steps need to be adjusted as well. We purpose two optional schemes for adjusting variable activities, **i-UIP-Inclusive** and **i-UIP-Exclusive**.

After learning C_1 from 1-UIP scheme, **i-UIP-Inclusive** collects all literals appear in the the **LBDstable-iUIP** clause C_i, and increase their variable activity uniformly according to the current branching heuristic if the literals' variable activity have not yet been bumped during 1-UIP. The scheme does not bump variable activities for transient literals that are resolved away at non-conflicting level because, unlike transient literals at conflicting level, these literals cannot be re-asserted after the immediate backtracking. Notice that other post-analyze extensions such as Reason Side Rate (RSR) and Locality[] can be applied after **i-UIP-Inclusive**.

$$\forall l \in C_i \cdot NotBumped(var(l)) \rightarrow bumpActivity(var(l))$$

i-UIP-Exclusive collects literals appear exclusively in C_i and bump their variable activity uniformly. It also find all the literals in C_1 that are resolved away during **LBDstable-iUIP**, and unbump their variable activity if they have been bumped during 1-UIP. The unbumped literals are no longer in the learned clause C_i, keep their variable activity bumped does not help solver to use C_i.

$$\forall l_i \in C_i \setminus C_1 \cdot bumpActivity(var(l_i))$$

$$\forall l_1 \in C_1 \setminus C_i \cdot unbumpActivity(var(l_1))$$

Algorithm 4 LBDstable-iUIP-CB

Require: C_1 is a valid 1-UIP clause
Require: $\mathcal{T}$ is a valid assertion trail
Require: lit_Order is a priority queue
1: **LBDSTABLE-iUIP-CB**($C_1, \mathcal{T}, lit\_Order$)
2: ...
3: $C_i \leftarrow C_1$ ▷ initialize i-UIP clause
4: forall $l \in C_1 \cdot$ Enqueu(lit_Order, l)
5: DecisionLvs $\leftarrow \{$ level(l) $\mid l \in C_1 \}$
6: **while** $lit\_Order \neq \emptyset$
7: $p \leftarrow$ dequeu(lit_Order, l)
8: **if** $\exists q \in$ Reason($\neg p, \mathcal{T}$) $\cdot$ level(q) $\notin$ DecisionLvs
9: $\vee p$ is the last reminaing lit in its decision level
10: Pass
11: **else**
12: $C_i \leftarrow (C_1) \bowtie$ Reason($q, \mathcal{T}$)
13: forall $l \in$ Reason($q, \mathcal{T}$) $\cdot$ Enqueu(lit_Order, l)
14: ...

3.5 Integrate with Chronological Backtracking

In SAT solver with chronological backtracking, literals on the assertion trail $\mathcal{T}$ are not always sorted by decision levels. This change imposes a challenge to **LBDstable-iUIP** learning since the previous implementation relies on solver's ability to efficiently access all literals from any decision level in descending trail order (alg. 1 line **??**).

To mitigate this challenge, we modify the solver to track the precise trail position of all asserted literals with a single vector. We then build a priority queue lit_Order to manage literal's resolution order. The queue lit_Order prioritizes literals with higher decision level, and it favors literal with higher trail position when decision levels are tied. The order of lit_Order represents the correct resolution order of **LBDstable-iUIP** because 1) an asserted literal's decision level is the maximum level among all of its reasoning literals, and 2) an asserted literal always appears higher in the trail than its reasoning literal.

Alg. 4 is the pesudo-code implementation of the augmented **LBDstable-iUIP** for chronological backtracking with the priority queue lit_Order. The algorithm first populates lit_Order with all literals in C_1 (line 4), and then continuously pops literals until the queue is empty (line 6). When a literal is resolved away, all of its reasoning literals are added into lit_Order if they are not already in the queue (line 13). The algorithm will always terminate because a literal cannot entered the queue twice (guaranteed by the properties of the trail order) and there are finite amount of literals on the trail.

4 Implementation and Experiments

4.1 Clause Reduction with LBDstable-iUIP

To evaluate **LBDstable-iUIP**'s effectiveness as a clause reduction technique, we implement **Pure-iUIP** and **Min-iUIP** with Control-I-UIP and Early Stopping rules in

Sec 3 on top of *MapleCOMSPS_LRB* [], the winner of SAT Race 2015 application track. We than compare the performance of the baseline *MapleCOMSPS_LRB* with Maple-**Pure-*i*UIP** and Maple-**Min-*i*UIP** on the full set of benchmarks from SAT RACE 2019 main track.

The benchmark contains 400 instances divided into two groups of 200, new and old, representing historical instances and fresh instances in the 2019 race, respectively. We randomly partition the old group instances into six partitions of size 30 and one partition of size 20. Each partition is then assigned to a XeonE5-2 CPU node with 16 cores (2 sockets 8 cores and 1 thread) and 96649 MB memory. The new group is partitioned based on their contributor (e.g. Heule contributed 22 matrix multiplication instances), and each partition is assigned to a aforementioned CPU node. To speed up the experiment, we allow a CPU node to solver at most seven instances concurrently.

Beside solved instances count and PAR-2 score, we additionally measure the average clause length and clause reduction ratio (both cumulative and non-cumulative)[4] for each instances. For Maple-**Pure-*i*UIP** and Maple-**Min-*i*UIP**, we also captures the **LBDstable-*i*UIP** learning attempted rate and success rate.

Solver	# solved	PAR-2	Clause Size	Cl Reduction%
MapleCOMSPS_LRB	221 (132, 89)	5018.89	62.6	36.53%
Maple-**Pure-*i*UIP**	**228** (135, 93)	4867.37	49.88	41.6% (42.72%)
Maple-**Min-*i*UIP**	226 (135, 91)	4890.67	**45.2**	**47.8% (51.19%)**

Fig. 2. Benchmark results of *MapleCOMSPS_LRB* , Maple-**Pure-*i*UIP** and Maple-**Min-*i*UIP** on SAT2019 race main track. CL Reduction% is the clause size reduction ratio comparing to non-minimized 1-UIP clauses, and the values in the brackets are the non-cumulative reduction ratio.

Fig. 2 shows that **Pure-*i*UIP** (**Pure-*i*UIP**) solved seven (five) more instances than the baseline solver with lower PAR-2 scores. **Pure-*i*UIP** has marginally lower PAR-2 score than **Pure-*i*UIP**. Both **Pure-*i*UIP** and **Min-*i*UIP** produce clause with significantly smaller size than 1-UIP by 20.4% and 27.7%, respectively. Fig. 3 shows the probability density distribution (PDF) of the relative clause size of both i-uip learning schemes (**Pure-*i*UIP** in red and **Min-*i*UIP** in green) for each instance. **Pure-*i*UIP** (**Min-*i*UIP**) produces shorter clauses for 77.7% (88.5%) of instances, and average relative reduction from 1-UIP is 16.7% (18.6%). Fig. 4 compares the absolute clause size of **Min-*i*UIP**, **Pure-*i*UIP** and 1-UIP, and it shows that both i-UIP learning schemes in general produce the smaller clauses, and the size reduction is more significant for instances with large average 1-UIP clause size. Moreover, **Min-*i*UIP** clauses (indicated in green) is consistently smaller than **Pure-*i*UIP** clauses.

2 Do we need this? We additionally looked at the 14 instances solved by **Min-*i*UIP**

[4] The cumulative reduction ratio is obtained through learning all clauses with the target learning scheme; Therefore, the reduction is cumulative. The non-cumulative reduction ratio is obtained by running the target scheme for measurement only (the minimized 1-UIP clause is learned); Therefore, the reduction is not cumulative.

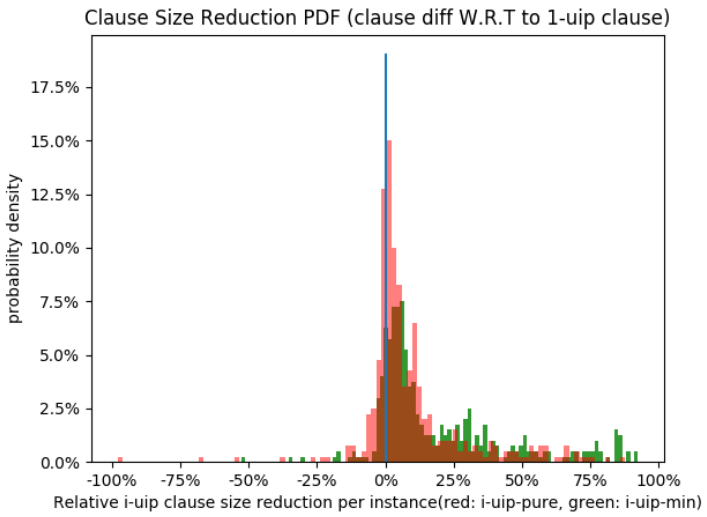

Fig. 3. Relative clause size reduction distribution. X axis indicates the relative size of difference between i-uip and 1-uip clauses (calculated as $\frac{|C_1| - |C_i|}{|C_1|}$) for each instance, and Y axis shows the probability density.

but not by 1-UIP. **Min-iUIP** produces smaller clauses for all of them with average relative reduction of 22% and maximum 77% (30 vs 135). Seven out of 14 instances has size relative reduction over 30%. For the 9 instances solved by 1-UIP but not by **Min-iUIP**, **Min-iUIP** only produce smaller clause for three instances and with average relative reduction of 3.3%.

Min-iUIP outperformed **Pure-iUIP** in clause size. This results agrees with our observation in Fig. 5: **Min-iUIP** attempted **LBDstable-iUIP** learning more frequently, and it is more likely to succeed.

A solver produce smaller clauses can construct smaller proofs. For UNSAT instances, we measure their DRUP[] proof checking time as well as the size of the optimized DRUP proof. We used DART-trim [] with 5000 timeout to check and optimize DRUP proofs.

Fig. 6 shows that the optimized proof construct by **Min-iUIP** and **Pure-iUIP** are significantly smaller than 1-UIP proofs. The relative proof size reduction roughly correlates to the average clause size reduction. Fig. 7 shows the absolute proof size comparison results.

4.2 LBDstable-iUIP as a Practical Learning Scheme

To evaluate **LBDstable-iUIP**'s effectiveness as a clause learning scheme, we re-implement **Min-iUIP** on *MapleCOMSPS_LRB* with the extensions mentioned in section 3. We evaluated 1-UIP learning and five **LBDstable-iUIP** configurations (**Min-iUIP**, **Pure-iUIP**,

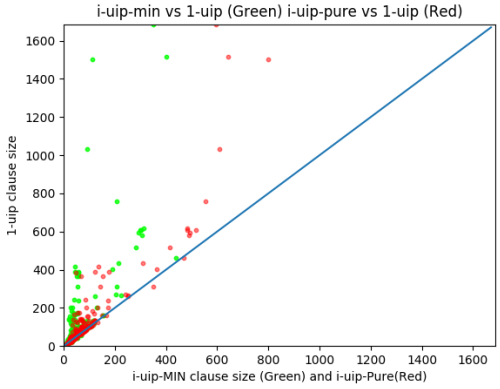

Fig. 4. Average clause size comparison plot. Each point in the plot represents an instance. X and Y axis shows the clause length from **LBDstable-iUIP** and 1-UIP, respectively. Each green (red) dot represents an compared instance between *MapleCOMSPS_LRB* and Maple-**Min-iUIP** (**Pure-iUIP**).

Solver	**LBDstable-iUIP** attempt rate	**LBDstable-iUIP** success rate
Maple-**Pure-iUIP**	16.1%	43.4%
Maple-**Min-iUIP**	28.8%	59.3%

Fig. 5. Compare **Pure-iUIP** and **Min-iUIP** i-uip attempt rate and success rate. **Min-iUIP** scheme attempted **LBDstable-iUIP** more frequently, and it is more likely to successfully produce smaller C_i clause .

Solver	optimized proof size (MB)	relative reduction size
MapleCOMSPS_LRB	613.9	0
Maple-**Pure-iUIP**	487.2	6.90%
Maple-**Min-iUIP**	413.2	17.18%

Fig. 6. Optimized UNSAT proof comparison for 1-UIP **Pure-iUIP** and **Min-iUIP**. Optimized proof size measures the average absolute proof size in MB, and relative reduction size measures the average relative reduction for all UNSAT instances.

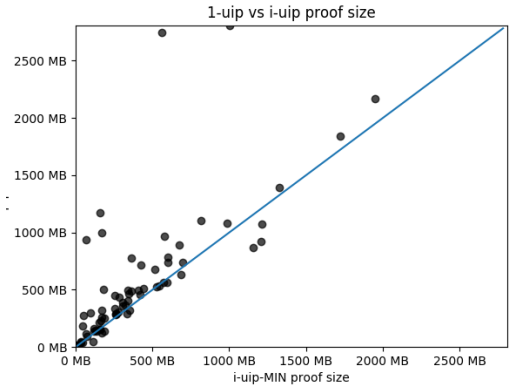

Fig. 7. Average optimized proof size between 1-uip and **Min-iUIP**.

Solver	# solved (SAT, UNSAT)	PAR-2	Avg clause Size
1-UIP	221 (132, 89)	5018.89	62.6
Pure-iUIP	**228** (135, 93)	4867.37	49.88
Min-iUIP	226 (135, 91)	4890.67	45.2
i-UIP-Greedy	226 (135, 91)	**4866.94**	47.7
i-UIP-Inclusive	225 (**138**, 87)	4958.49	52..12
i-UIP-Exclusive	223 (134, 89)	5015.23	**43.2**

Fig. 8. Benchmark results of *MapleCOMSPS_LRB* with 1-UIP, **Pure-iUIP**, **Min-iUIP**, **i-UIP-Greedy**, **i-UIP-Inclusive**, and **i-UIP-Exclusive** on SAT2019 race main track.

i-UIP-Greedy, **i-UIP-Inclusive**, and **i-UIP-Exclusive**) on the SAT Race 2019 main track benchmark and reported solved instances, PAR-2 score and average clause size.

Fig. 8 summarizes the result of the experiment. Learning scheme **Pure-iUIP** solved the most overall instances (228) and the most UNSAT instances (93). **i-UIP-Greedy** had the lowest PAR-2 score. **i-UIP-Inclusive** solved the most SAT instances (138). **i-UIP-Exclusive** produced the shortest average clause size, but solved the second least instances, one more instance than the baseline 1-UIP learning. All configurations of **LBDstable-iUIP** outperformed the baseline 1-UIP scheme in solved instances, PAR-2 score and average clause size.

4.3 LBDstable-iUIP on Modern SAT solvers

To validate **LBDstable-iUIP** as a generalizable learning scheme on modern SAT solvers, we re-implement **LBDstable-iUIP** on the winners of 2017, 2018 and 2019 SAT Race[] and *expMaple_CM_GCBumpOnlyLRB*[] (*expMaple*). *expMaple* is a top ten solver from 2019 SAT race which uses random walk simulation to help branching. We chose *expMaple* because 1) it is a top solver in the 2019 SAT Race without using chronological backtracking; 2) the combination of random walk simulation and variable activity branching heuristic allows our learning schemes to partially sidestep the problem of variable activity. For each solver, we compare the base 1-UIP learning scheme against **Pure-iUIP**, **Min-iUIP** and the top two **LBDstable-iUIP** variants, **i-UIP-Greedy** and **i-UIP-Inclusive**, on the SAT Race 2019 main track benchmark. We report solved instances, PAR-2 score and the average clause size.

Table 9 shows the benchmark result of **LBDstable-iUIP** configurations on different solvers. All four configurations of **LBDstable-iUIP** outperformed 1-UIP learning for the SAT 2017 race winner, *MapleLCMDist*. More specifically, **Pure-iUIP**, **Min-iUIP**, **i-UIP-Greedy** and **i-UIP-Inclusive** solved 12, 8, 5 and 2 more instances, respectively, whiling producing smaller clauses. **Pure-iUIP** solved more UNSAT and SAT instances while other configurations improved on solving SAT instances. The improvement of **LBDstable-iUIP** is more significant on *MapleLCMDist* than on *MapleCOMSPS_LRB* for both solved instances and clause size reduction. This may suggests that **LBDstable-iUIP** and the recent learnt clause minimization approach [] synergies well because i-UIP clauses are shorter with more common literals which allows vivification [] to prune literals more aggressively through unit propagation.

Solver	# solved (SAT, UNSAT)	PAR-2	Avg clause Size
SAT 2017 Winner *MapleLCMDist*	232 (135, 97)	4755.96	61.9
MapleLCMDist-i-pure	**244 (146, 98)** +12	**4504.18**	43.76
MapleLCMDist-i-min	240 (144, 96) +8	4601.25	**36.97**
MapleLCMDist-i-greedy	237 (140, 97) +5	4678.434	43.62
MapleLCMDist-i-inclusive	234 (137, 97) +2	4718.03	37.96
SAT 2018 Winner *MapleCB*	236 (138, 98)	4671.81	61.69
MapleCB-i-pure	**241** (**142, 99**) +5	**4598.18**	44.19
MapleCB-i-min	236 (141, 95) +0	4683.92	38.05
MapleCB-i-greedy	240 (141, **99**) +4	4626.99	41.16
MapleCB-i-inclusive	240 (**142**, 98) +4	4602.13	**37.52**
SAT 2019 Winner *MapleCB-DL*	238 (140, **98**)	4531.24	60.91
MapleCB-DL-i-pure	238 (140, **98**) +0	4519.08	43.32
MapleCB-DL-i-min	244 (**148**, 96) +6	**4419.84**	**36.88**
MapleCB-DL-i-greedy	243 (146, 97) + 5	4476.73	40.65
MapleCB-DL-i-inclusive	**243** (**148**, 95) +5	4455.76	37.02
SAT 2019 Competitor *expMaple*	237 (137, 100)	4628.96	63.19
expMaple-i-pure	235 (136, 99) -2	4668.96	48.26
expMaple-i-min	241 (143, 98) +4	4524.28	46.29
expMaple-i-greedy	244 (143, **101**) +7	**4460.92**	47.25
expMaple-i-inclusive	**245** (**146**, 99) +8	4475.76	**45.33**

Fig. 9. Benchmark results of 1-UIP, **Pure-iUIP**. **Min-iUIP**, **i-UIP-Greedy** and **i-UIP-Inclusive** on SAT2019 race main track.

We observed significant improvement of **LBDstable-iUIP** for the SAT 2018 winner, *MapleCB*. Three out of four **LBDstable-iUIP** configurations outperformed 1-UIP by 5, 4 an 4 instances, respectively. *MapleCB* improved from *MapleLCMDist* by using chronological backtracking (CB) for long distance backtracks. Therefore, we used **LBDstable-iUIP**-CB extension for all **LBDstable-iUIP** configurations. The results indicates that the improvement of **LBDstable-iUIP** is slightly shadowed by the adoption of CB. More specifically, we believe CB prevents decision levels from being compressed through the process of backtracking and literal assertion. The shorter i-UIP clauses can bring related literals closer, which in turns compresses the assertion stack and the decision levels. However, since CB discourages long distance backtracking, the effect of learning shorter clauses is shadowed until a full restart. We believe we have observed an interesting interaction effect that shows the limitations of both CB and **LBDstable-iUIP** learning for future research.

LBDstable-iUIP learning schemes showed significant improvement for *MapleCB-DL*. Three out of four configurations improved solved instances by 6, 5 and 5 instances, respectively. *MapleCB-DL* uses duplicate learning (DL) to prioritize clauses that are learned multiple times. **LBDstable-iUIP** and DL didn't synergies well possibly because i-UIP clauses are less likely to be duplicated. We observed that **Min-iUIP** ,in average, added 12% less duplicated clauses into the core clause database than 1-UIP. One possible explanation is that **LBDstable-iUIP** learning can reduce an 1-UIP clause to different i-UIP clauses for different assertion trails. Therefore, the solver is less likely to learn the same i-UIP clauses.

We observed significant improvement of **LBDstable-iUIP** for *expMaple*, three out of four configurations of **LBDstable-iUIP** significantly outperformed 1-UIP learning by 4, 7 and 8 more instances, respectively, while producing smaller clauses. We expect all three configurations of **LBDstable-iUIP** to solve the similar amount of instances with close PAR-2 scores because the additional random walk exploration allows the learning schemes to partially sidestep the activity problem; hence, the learning adjustment for variable activities should have less impact on the solver's performance. However, we instead, observed that both **i-UIP-Greedy** and **i-UIP-Inclusive** outperformed the default i-UIP learning scheme. One possible explanation is that **i-UIP-Greedy** and **i-UIP-Inclusive** schemes could easily overcompensate variable activities for i-UIP clauses, and *expMaple*'s random walk exploration could use future search information to mitigate the negative effects of our overcompensation.

References

1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009. pp. 399–404 (2009), http://ijcai.org/Proceedings/09/Papers/074.pdf
2. Beame, P., Kautz, H.A., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. J. Artif. Intell. Res. **22**, 319–351 (2004), https://doi.org/10.1613/jair.1410
3. Jr., R.J.B., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: Kuipers, B., Webber, B.L. (eds.) Proceedings of the Fourteenth National Conference

on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA. pp. 203–208. AAAI Press / The MIT Press (1997), http://www.aaai.org/Library/AAAI/1997/aaai97-032.php
4. Katebi, H., Sakallah, K.A., Silva, J.P.M.: Empirical study of the anatomy of modern sat solvers. In: Sakallah, K.A., Simon, L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6695, pp. 343–356. Springer (2011), https://doi.org/10.1007/978-3-642-21581-0_27
5. Knuth, D.E.: Implementation of algorithm 7.2.2.2c (conflict-driven clause learning sat solver), https://www-cs-faculty.stanford.edu/~knuth/programs/sat13.w
6. Luo, M., Li, C., Xiao, F., Manyà, F., Lü, Z.: An effective learnt clause minimization approach for CDCL SAT solvers. In: Sierra, C. (ed.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017. pp. 703–711. ijcai.org (2017), https://doi.org/10.24963/ijcai.2017/98
7. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001. pp. 530–535. ACM (2001), https://doi.org/10.1145/378239.379017
8. Pipatsrisawat, K., Darwiche, A.: On the power of clause-learning SAT solvers as resolution engines. Artif. Intell. **175**(2), 512–525 (2011), https://doi.org/10.1016/j.artint.2010.10.002
9. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 131–153. IOS Press (2009), https://doi.org/10.3233/978-1-58603-929-5-131
10. Silva, J.P.M., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: Rutenbar, R.A., Otten, R.H.J.M. (eds.) Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996. pp. 220–227. IEEE Computer Society / ACM (1996), https://doi.org/10.1109/ICCAD.1996.569607
11. Silva, J.P.M., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Computers **48**(5), 506–521 (1999), https://doi.org/10.1109/12.769433
12. Sörensson, N., Biere, A.: Minimizing learned clauses. 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. 237–243. Springer (2009), https://doi.org/10.1007/978-3-642-02777-2_23
13. Wieringa, S., Heljanko, K.: Concurrent clause strengthening. In: Järvisalo, M., Gelder, A.V. (eds.) Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7962, pp. 116–132. Springer (2013), https://doi.org/10.1007/978-3-642-39071-5_10
14. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: Ernst, R. (ed.) Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001. pp. 279–285. IEEE Computer Society (2001), https://doi.org/10.1109/ICCAD.2001.968634