Contribution Title*

First Author[1][0000−1111−2222−3333] and Fahiem Bacchus[2,3][1111−2222−3333−4444]

[1] Princeton University, Princeton NJ 08544, USA
[2] Springer Heidelberg, Tiergartenstr. 17, 69121 Heidelberg, Germany
lncs@springer.com
http://www.springer.com/gp/computer-science/lncs
[3] ABC Institute, Rupert-Karls-University Heidelberg, Heidelberg, Germany
{abc,lncs}@uni-heidelberg.de

1 Introduction

2 Preliminaries

3 Find i-uip Clause

Classical CDCL SAT solvers analyze each conflict with 1-UIP resolution learning scheme to derive an asserting clause C_1. Even though prior studies shows that further resolutions on C_1 against the assertion trail ϕ cannot reduce the clause's literal block distance (LBD), the resolutions could potentially reduce the clause's size. Clause size is an important quality measurement because a smaller clause 1) consumes less memory, 2) requires less steps to force a literal and 3) decreases the size of future conflict clauses.

Inspired by both LBD and clause size, i-UIP resolution learning attempts to resolve away literals in C_1 against the assertion trail ϕ to minimize the clause size without increasing the clause's LBD. The goal of i-UIP learning is to find a clause C_i whose literals are either the unique implication points in their respective decision levels or are directly implied by literals from a foreign decision level. Alg. 1 is a pseudo-code implementation of i-UIP learning.

The algorithm **i-UIP** computes C_i from (line 2 to 20), and then returns the smaller clause between C_1 and C_i (line 26 to 6). The algorithm first initializes C_i with the minimized[] C_1. Next, the algorithm iterate through the decision levels of C_i in descending order (line 4) and tries to find the unique implication point in each level (line 6). Since **i-UIP** needs to preserve C_i's LBD during resolutions, before resolving away a target literal p, the algorithm preemptively checks whether the reasoning clause of $\neg p$ contains any literal q from an unseen decision level (line 9). If such a literal q exists, then the algorithm cannot find the unique implication point (UIP) at the current level i because resolving away p will introduce q into the clause and increases LBD. One solution is to abandon level i by reverting back to the state before any literal at level i is resolved away (line 10), and then move on to the next level (line 11). We denote **i-UIP** learning with this solution as **Pure-i-UIP**. The final C_i produced by **Pure-i-UIP**

* Supported by organization x.

Algorithm 1 i-UIP

Require: C_1 is a valid and minimized 1-UIP clause
Require: ϕ is a valid assertion trail
1: **procedure i-UIP**(C_1, ϕ)
2: $C_i \leftarrow C_1$ ▷ initialize i-UIP clause
3: DecisionLvs $\leftarrow$ Decision levels in C_1 in descending order
4: **for** $i \in$ DecisionLvs **do**
5: $L_i \leftarrow \{\, l \mid \text{level}(l) = i \wedge l \in C_i \,\}$
6: **while** $|\text{unmarked}(L_i)| > 1$ **do**
7: $p \leftarrow$ lit with the highest trail position in L_i
8: **if** $\exists q \in \text{Reason}(\neg p, \phi) \cdot \text{level}(q) \notin$ DecisionLvs **then**
9: **if Pure-i-UIP then**
10: Unresolve literals at level i
11: Go to the next decision level i
12: **else if Min-i-UIP then**
13: Mark(p)
14: **end if**
15: **else**
16: $C_i \leftarrow (C_1) \bowtie \text{Reason}(\neg p, \phi)$
17: Update(L_i)
18: **end if**
19: **end while**
20: **end for**
21:
22: **if Pure-i-UIP then** ▷ C_i from **Pure-i-UIP** can be minimized
23: $C_i \leftarrow$ **Minimize**(C_i)
24: **end if**
25:
26: **if** $|C_i| < |C_1|$ **then**
27: **return** C_i
28: **else**
29: **return** C_1
30: **end if**
31: **end procedure**

will contains exactly one literal for levels where the LBD-persevering UIP exist. However, **Pure-i-UIP** does not minimize the number of literals in the decision levels where the LBD-preserving UIP does not exist, and clause minimization techniques[] can be used to further reduce the clause size (line 22 to 24).

When **i-UIP** cannot resolve away a literal p without increasing C_i's LBD, instead of skipping the entire decision level i, another solution **Min-i-UIP** simply marks literal p and keep it in C_i (line 13). The learning scheme then continue the resolutions at level i. **Min-i-UIP** will ignore all the unsolvable literals and find the "local" unique implication point at every decision level. The algorithm terminates when there is exactly one unmarked literal left for each decision level

of C_i, representing the local unique implication points. The clause C_i learnt by **Min-i-UIP** is minimized[4], we defer the proof of minimization to Appendix **??**.

Example 1. Consider the assertion trail $\phi = ..a_2, b_2, c_2, d_2..e_5, f_5, g_5, h_6, i_6, j_6, k_6..$ where literals $a_2...k_6$ are asserted in order with decision level shown as the underscript. Let L_{ext} denote the set of literals on ϕ outside of decision level 2,5,6 and 10, and l^*_{ext} be some literal in L_{ext}. The reasoning clauses involved in building ϕ are:

$C_k = \{\neg f_5 \vee \neg j_6 \vee k_6\}$, $C_j = \{\neg f_5 \vee \neg i_6 \vee j_6\}$, $C_i = \{\neg e_5 \vee \neg h_6 \vee i_6\}$, $C_g = \{\neg a_2 \vee \neg f_5 \vee g_5\}$ $C_f = \{\neg l^*_{ext} \vee \neg e_5 \vee f_5\}$, $C_d = \{\neg b_2 \vee \neg c_2 \vee d_2\}$, $C_c = \{\neg a_2 \vee \neg b_2 \vee c_2\}$, $C_b = \{\neg l^*_{ext} \vee \neg a_2 \vee b_2\}$

Suppose 1-UIP learning produces clause $C_1 = \{\neg m_{10} \vee \neg k_6 \vee \neg j_6 \vee \neg i_6 \vee \neg h_6 \vee \neg g_5 \vee \neg d_2 \vee \neg c_2\}$ where $\neg k_{10}$ is the UIP from the conflicting level. **i-UIP** first tries to find the UIP for level 6 by resolving C_1 with C_k, C_j and, C_i in order and produces clause $C_i = \{\neg m_{10} \vee \neg h_6 \vee \neg g_5 \vee \neg f_5 \vee \neg e_5 \vee \neg d_2 \vee \neg c_2\}$ where $\neg h_6$ is the UIP.

i-UIP attempts to find the UIP for level 5 by resolving C_i with C_g and C_f in order. However, resolving with C_f will introduce $\neg l^*_{ext}$ into C_i and increase the clause's LBD. **Pure-i-UIP** undoes all resolutions at level 5 and skips the entire level, whereas **Min-i-UIP** only skips the resolution with C_f and tries to find the UIP by ignoring $\neg f_5$. After level 5, **Pure-i-UIP** preserves the same clause and **Min-i-UIP** produces $C_i = \{\neg m_{10} \vee \neg h_6 \vee \neg f_5 \vee \neg e_5 \vee \neg d_2 \vee \neg c_2 \vee \neg a_2\}$ where $\neg e_5$ is the UIP when $\neg f_5$ is ignored. The same **i-UIP** procedure is then performed on level 2 to produce the final C_i.

Pure-i-UIP learned $C_i = \{\neg m_{10} \vee \neg h_6 \vee \neg g_5 \vee \neg f_5 \vee \neg e_5 \vee \neg d_2 \vee \neg c_2\}$ and reduces the learnt clause size by 1. **Min-i-UIP** learned $C_i = \{\neg m_{10} \vee \neg h_6 \vee \neg f_5 \vee \neg e_5 \vee \neg b_2 \vee \neg a_2\}$ and reduces the learnt clause size by 2.

The core algorithm of **i-UIP** 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 **i-UIP** as a practical clause learning scheme.

3.1 Control i-UIP Learning

This simple and greedy **i-UIP** learning scheme can produce significantly smaller clause. However, when **i-UIP** 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 **i-UIP** 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 **i-UIP** is unlikely to achieve cost effective results. Therefore, we propose a heuristic based approach to to enable and disable **i-UIP** learning based on input clause's Gap.

[4] clause size cannot be reduced via minimization techniques[].

Algorithm 2 Control-**i-UIP**

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

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 **i-UIP** learning.

The gap threshold t_{gap} is initialized to 0, and is updated at every restart based on **i-UIP**'s success rate from the previous restart interval. More specifically, the algorithm collects the statistics of the number of **i-UIP** learning attempted (line 9 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 **i-UIP**for the next restart interval. On the other hand, the threshold is decreased to encourage more aggressive **i-UIP** 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-i-UIP** 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 **i-UIP** 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 **i-UIP** has already failed.

The early stopping rule may prematurely stops **Pure-i-UIP** 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.

Algorithm 3 i-UIP-Greedy

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

3.3 Greedy Active Clause Selection

Even though **i-UIP** 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 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 **i-UIP** 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 **i-UIP** 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**.

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

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 **i-UIP**, and unbump their variable activity if they have been bumped

Algorithm 4 i-UIP-CB

Require: C_1 is a valid 1-UIP clause
Require: ϕ is a valid assertion trail
Require: *lit_Order* is a priority queue
1: **procedure i-UIP**-CB(C_1, ϕ, 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 $\leftarrow \{\, \text{level}(l) \mid l \in C_1 \,\}$
6: **while** $lit\_Order \neq \emptyset$ **do**
7: $p \leftarrow$ dequeu(lit_Order, l)
8: **if** $\exists q \in \text{Reason}(\neg p, \phi) \cdot \text{level}(q) \notin$ DecisionLvs
9: $\vee p$ is the last reminaing lit in its decision level **then**
10: Pass
11: **else**
12: $C_i \leftarrow (C_1) \bowtie \text{Reason}(q, \phi)$
13: forall $l \in \text{Reason}(q, \phi) \cdot$ Enqueu(lit_Order, l)
14: **end if**
15: **end while**
16: ...
17: **end procedure**

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.

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

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

3.5 Integrate with Chronological Backtracking

In SAT solver with chronological backtracking, literals on the assertion trail ϕ are not always sorted by decision levels. This change imposes a challenge to **i-UIP** 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 7).

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 **i-UIP** 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 **i-UIP** 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 i-UIP

To evaluate **i-UIP**'s effectiveness as a clause reduction technique, we implement **Pure-i-UIP** and **Min-i-UIP** 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)[5] for each instances. For Maple-**Pure-i-UIP** and Maple-**Min-i-UIP**, we also captures the **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. 1: 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.

[5] 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. 1 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. 2 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. 3 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.

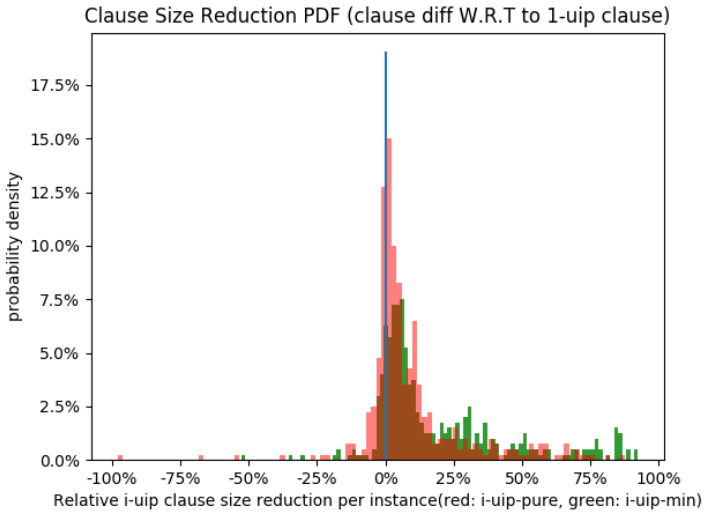

Fig. 2: Relative clause size reduction distribution. X axis indicates the relative size of difference between i-uip and 1-uip clauses (calculated as $\frac{|C_1| - |C_i|}{|C_1|}$) for each instance, and Y axis shows the probability density.

Do we need this? We additionally looked at the 14 instances solved by **Min-i-UIP** but not by 1-UIP. **Min-i-UIP** 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-i-UIP**, **Min-i-UIP** only produce smaller clause for three instances and with average relative reduction of 3.3%.

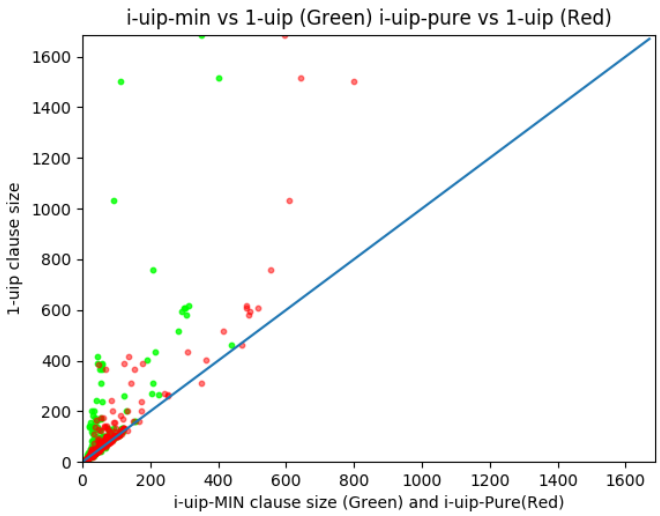

Fig. 3: Average clause size comparison plot. Each point in the plot represents an instance. X and Y axis shows the clause length from **i-UIP** and 1-UIP, respectively. Each green (red) dot represents an compared instance between *MapleCOMSPS_LRB* and Maple-**Min-i-UIP** (**Pure-i-UIP**).

Min-i-UIP outperformed **Pure-i-UIP** in clause size. This results agrees with our observation in Fig. 4: **Min-i-UIP** attempted **i-UIP** learning more frequently, and it is more likely to succeed.

Solver	**i-UIP** attempt rate	**i-UIP** success rate
Maple-**Pure-i-UIP**	16.1%	43.4%
Maple-**Min-i-UIP**	28.8%	59.3%

Fig. 4: Compare **Pure-i-UIP** and **Min-i-UIP** i-uip attempt rate and success rate. **Min-i-UIP** scheme attempted **i-UIP** more frequently, and it is more likely to successfully produce smaller C_i clause .

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. 5 shows that the optimized proof construct by **Min-i-UIP** and **Pure-i-UIP** are significantly smaller than 1-UIP proofs. The relative proof size reduction

roughly correlates to the average clause size reduction. Fig. 6 shows the absolute proof size comparison results.

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

Fig. 5: Optimized UNSAT proof comparison for 1-UIP **Pure-i-UIP** and **Min-i-UIP**. Optimized proof size measures the average absolute proof size in MB, and relative reduction size measures the average relative reduction for all UNSAT instances.

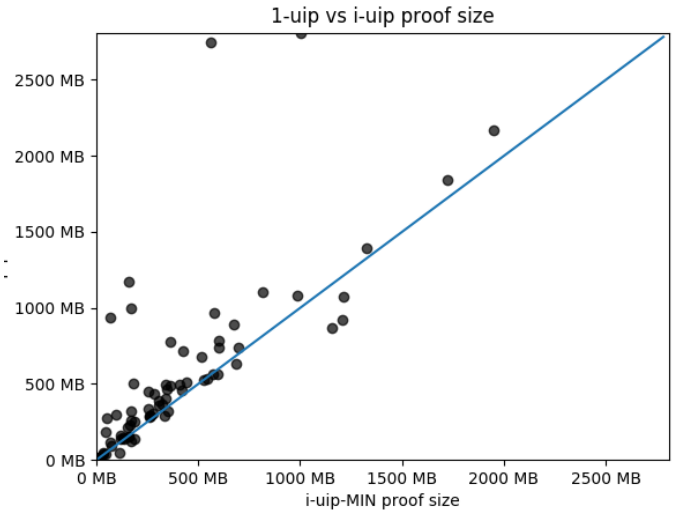

Fig. 6: Average optimized proof size between 1-uip and **Min-i-UIP**.

4.2 i-UIP as a Practical Learning Scheme

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

Solver	# solved (SAT, UNSAT)	PAR-2	Avg clause Size
1-UIP	221 (132, 89)	5018.89	62.6
Pure-i-UIP	**228** (135, 93)	4867.37	49.88
Min-i-UIP	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. 7: Benchmark results of *MapleCOMSPS_LRB* with 1-UIP, **Pure-i-UIP**, **Min-i-UIP**, **i-UIP-Greedy**, **i-UIP-Inclusive**, and **i-UIP-Exclusive** on SAT2019 race main track.

2019 main track benchmark and reported solved instances, PAR-2 score and average clause size.

Fig. 7 summarizes the result of the experiment. Learning scheme **Pure-i-UIP** 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 **i-UIP** outperformed the baseline 1-UIP scheme in solved instances, PAR-2 score and average clause size.

4.3 i-UIP on Modern SAT solvers

To validate **i-UIP** as a generalizable learning scheme on modern SAT solvers, we re-implement **i-UIP** 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-i-UIP**, **Min-i-UIP** and the top two **i-UIP** 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 8 shows the benchmark result of **i-UIP** configurations on different solvers. All four configurations of **i-UIP** outperformed 1-UIP learning for the SAT 2017 race winner, *MapleLCMDist*. More specifically, **Pure-i-UIP**, **Min-i-UIP**, **i-UIP-Greedy** and **i-UIP-Inclusive** solved 12, 8, 5 and 2 more instances, respectively, whiling producing smaller clauses. **Pure-i-UIP** solved more UNSAT and SAT instances while other configurations improved on solving SAT instances. The improvement of **i-UIP** is more significant on *MapleLCMDist* than on *MapleCOMSPS_LRB* for both solved instances and clause size reduction. This may suggests that **i-UIP** and the recent learnt clause minimization

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

approach [] synergies well because i-UIP clauses are shorter with more common literals which allows vivification [] to prune literals more aggressively through unit propagation.

We observed significant improvement of **i-UIP** for the SAT 2018 winner, *MapleCB*. Three out of four **i-UIP** 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 **i-UIP**-CB extension for all **i-UIP** configurations. The results indicates that the improvement of **i-UIP** 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 **i-UIP** learning for future research.

i-UIP 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. **i-UIP** and DL didn't synergies well possibly because i-UIP clauses are less likely to be duplicated. We observed that **Min-i-UIP** ,in average, added 12% less duplicated clauses into the core clause database than 1-UIP. One possible explanation is that **i-UIP** 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 **i-UIP** for *expMaple*, three out of four configurations of **i-UIP** significantly outperformed 1-UIP learning by 4, 7 and 8 more instances, respectively, while producing smaller clauses. We expect all three configurations of **i-UIP** 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. LNCS Homepage, http://www.springer.com/lncs. Last accessed 4 Oct 2017