Contribution Title*

First Author[1][0000−1111−2222−3333] and Second Author[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 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 21 to 7). 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** 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.

* 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: **if** $|C_i| < |C_1|$ **then**
22: **return** C_i
23: **else**
24: **return** C_1
25: **end if**
26: **end procedure**

When **i-UIP** cannot resolve away a literal p without increasing C_i's LBD, instead of skipping the entire decision level i, a more practical solution **Min-i-UIP** will mark and keep p in C_i (line 13), and continue 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 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.

1.1 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

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: **if** $|C_i| < |C_1|$ **then**
6: Succeed $\leftarrow$ Succeed + 1
7: **end if**
8: Attempted $\leftarrow$ Attempted + 1
9: **end if**
10: **end procedure**

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.

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 8 in alg. 2) and the number of attempts succeed (line 6) 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.

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

1.2 Early stop i-UIP Learning

At any point of **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.

Algorithm 3 i-UIP-Greedy

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

1.3 Greedy Active Clause Selection

Even though **i-UIP** learning can reduce the size of the learnt clause C_i, but it may introduce inactive literals 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 3). The algorithm learns C_i if the clause has smaller size and higher average variable activity.

1.4 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 applying **i-UIP-Inclusive** on C_i so that no literal's variable activity is double bumped.

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

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

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

during **i-UIP**, 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.

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

1.5 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.

References

1. LNCS Homepage, http://www.springer.com/lncs. Last accessed 4 Oct 2017