

Clause size reduction with all-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 [23,17,18,3]. 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 [23] employed in Chaff [12] (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 [19], 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 [23]. 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 all-UIP scheme [23]. 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 all-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 minimization methods that try to remove redundant literals from the 1-UIP clause yielding a clause that is a subset of the 1-UIP clause, e.g., [19,8,22]. Our method is orthogonal to clause minimization. In particular, our approach can learn a clause that is not a subset of 1-UIP clause but which still serves all of the same purposes as the 1-UIP clause. Various minimization techniques can be applied on top of our method to remove redundant literals from the clauses we learn.

We present various versions of our method and show that these variants are often capable of learning shorter clauses than 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 [18], 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 $\text{var}(\ell)$ denote its underlying variable. A CNF consists of a conjunction of clauses, each of 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 resolvent $C_1 \boxtimes C_2$ is the new clause $(C_1 \cup C_2) - \{\ell, \neg \ell\}$. The resolvent will be a tautology (i.e., a clause containing a literal l and its negation $\neg l$) 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 [16].

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, i.e. $\mathcal{T}[\iota(\ell)] = \ell$. For convenience, we also let $\iota(\ell) = \iota(\neg \ell) = \iota(\text{var}(\ell))$ even though neither $\neg \ell$ nor $\text{var}(\ell)$ 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.

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 remaining unfalsified literal of a clause. Each literal $\ell \in \mathcal{T}$ has a decision level $\text{decLvl}(\ell)$ which is equal to the number of decision literals appearing before ℓ on the trail plus one; hence, $\text{decLvl}(d) = 1$ 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 $\text{decLvl}(d_i) = i$ we call this subsequence of \mathcal{T} the *i -th decision level*.

Each literal $\ell \in \mathcal{T}$ also has a clausal reason $\text{reason}(\ell)$. If ℓ is a unit propagated literal, $\text{reason}(\ell)$ is a clause of the formula such that $\ell \in \text{reason}(\ell)$ and $\forall x \in \text{reason}(\ell). x \neq \ell \rightarrow (\neg x \in \mathcal{T} \wedge \iota(\neg x) < \iota(\ell))$. That is, $\text{reason}(\ell)$ 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 $\text{reason}(\ell) = \emptyset$.

In most SAT solvers 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. 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 learnt. After the new clause is learnt 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 $\mathcal{T}[0] \dots \mathcal{T}[\iota(d_k) - 1]$. 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 $\text{decLvl}(x) \leq \max\{\text{decLvl}(y) | y \in C_u \wedge y \neq x\}$. This means that if x appears in a clause made unit it must have been added to the trail, and added at or before decision level the clause became unit. Note that more than one clause might be made unit by \mathcal{T} forcing x , or x might be set as a decision before being forced. This condition ensures that x appears in \mathcal{T} at or before the first decision level it is forced by any clause.

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 Resolvent) *A trail resolvent is a clause arising from resolving a conflict against the reason clause of some literal $\ell \in \mathcal{T}$. Every trail resolvent is also a conflict.*

The following things can be noted about trail resolvents: (1) trail resolvents are never tautological, as the polarity of all literals in $\text{reason}(\ell)$ 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 $\text{var}(\ell)$ resolved on must be a unit propagated literal whose negation appears in the conflict; and (3) any variable in the conflict that is unit propagated in \mathcal{T} can be resolved upon (the 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 resolvents 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, \dots, v_m resolved have strictly decreasing trail indices: $\iota(v_{i+1}) < \iota(v_i)$ ($1 \leq i < m$). (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 loss 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 $C_I = C_0, C_1, \dots, C_m = C_L$ obtained by resolving on the sequence of variables v_1, \dots, v_m whose corresponding literals on \mathcal{T} are l_1, \dots, 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 indices, O must be a valid trail resolution if U was, and furthermore O yields the clause $C'_L = \bigcup_{i=1}^m \text{reason}(l_i) - \{l_1, \neg l_1, \dots, l_m, \neg l_m\}$. Since U resolves on the same variables (in a different order) using the same reason clauses we must have $C'_L \subseteq C_L$. It can, however, be the case that C'_L is proper subset of C_L : if l_i is resolved away it might be reintroduced when resolving on l_{i+1} if $\iota(l_{i+1}) > \iota(l_i)$. \square

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 [19] 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 [18] 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., [23,17,18,3]. 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 than the all-decision clause. Furthermore, it is known [18] 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 which stop at the last UIP of each level.

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

An **asserting clause** [14] is a conflict clause C_L that has exactly one literal ℓ at its deepest level, i.e., $\forall x \in C_L. \text{decLvl}(x) \leq \text{decLvl}(\ell) \wedge (\text{decLvl}(x) = \ell \rightarrow x = \ell)$. 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 [14], 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 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

<p>(a) All Decision Clause all_decision(C_I) $C \leftarrow C_I$ while $\{l \mid l \in C \wedge \text{reason}(l) \neq \emptyset\} \neq \emptyset$ $\ell \leftarrow$ literal with highest trail index in $\{l \mid l \in C \wedge \text{reason}(l) \neq \emptyset\}$ $C \leftarrow C \bowtie \text{reason}(\neg \ell)$ return C</p>	<p>(b) Make level i contain a single literal UIP_level(C, i) while $\left \left\{ \ell \mid \begin{array}{l} \ell \in C \\ \wedge \text{reason}(\ell) \neq \emptyset \\ \wedge \text{decLvl}(\ell) = i \end{array} \right\} \right > 1$ $\ell \leftarrow$ literal with highest trail index in $\left\{ \ell \mid \begin{array}{l} \ell \in C \wedge \text{reason}(\ell) \neq \emptyset \\ \wedge \text{decLvl}(\ell) = i \end{array} \right\}$ $C \leftarrow C \bowtie \text{reason}(\neg \ell)$ return C</p>
<p>(c) First UIP Clause 1-UIP(C_I) $i \leftarrow \max\{\text{decLvl}(l) \mid l \in C_I\}$ return UIP_level(C_I, i)</p>	<p>(e) i-UIP Clause i-UIP(C_I, i) $C \leftarrow C_I$ $d \leftarrow \max\{\text{decLvl}(l) \mid l \in C\}$ for ($j \leftarrow 1$; $j \leq i$; $j \leftarrow j + 1$) if ($d = \emptyset$): break $C \leftarrow$ UIP_level(C, d) $d \leftarrow \max \left\{ \text{decLvl}(l) \mid \begin{array}{l} l \in C \\ \wedge \text{decLvl}(l) < d \end{array} \right\}$ return C <i>Maximum of an empty set is \emptyset</i></p>
<p>(d) All UIP Clause all-UIP(C_I, i) $i = \{\text{decLvl}(l) \mid l \in \mathcal{T}\}$ $\triangleright i$ is large enough to ensure all levels are UIP return i-UIP(C, i)</p>	

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.

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. If two of its literals at the deepest level 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: its 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: $\text{LBD}(C_L) = |\{\text{decLvl}(l) \mid l \in C_L\}|$ [1]. Empirically LBD is a 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 among all asserting clauses that can be learnt from C_I . If C' is a trail resolvent of C and a reason clause $\text{reason}(l)$, then $\text{LBD}(C') \geq \text{LBD}(C)$ since $\text{reason}(l)$ must contain at least one other literal with the same decision level as l and might contain literals with decision levels not in C . That is, each trail resolution step can only increase the LBD of the learnt clause. Hence, the 1-UIP scheme yields an asserting clause with minimum LBD as it performs the minimum number of trail resolutions required to generate an asserting clause.

The other schemes must perform more trail resolutions. In fact, all of these schemes (all-decision, all-uip, i-uip) use trail resolutions in which the 1-UIP clause appears. That is, they all must first generate the 1-UIP clause and then continue with further

trail resolution steps. These extra resolution steps can introduce many additional decision levels into the final clause. Hence, these schemes yield learnt clauses whose LBD can be larger, and never smaller, than the 1-UIP clause.

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

3 Using all-UIP Clause Learning

Although learning clauses with low LBD has been shown empirically to be important in SAT solving than learning short clauses [1], clause size is still 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 all-UIP scheme will tend to produce small clauses since the clauses contain at most one literal per decision level. However, the all-UIP clause can have much higher LBD. Since, LBD is important than size our approach is to use all-UIP learning when, and only when, it succeeds in reducing the size of the clause *without increasing its LBD*. The all-UIP scheme first computes the 1-UIP clause when it reduces the deepest level to a single UIP literal. It then proceeds to reduce the shallower levels (see all-UIP's for loop in Figure 1). So our approach will start with the 1-UIP clause and then try to apply all-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 [19] 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 *stable-allUIP* 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, $decLvl[0]$, can be skipped. The subroutine *try-uiip-level* is used to reduce each level. It uses trail resolutions to achieve this, subject to the constraint that no new decision levels can be introduced in the clause. In particular, *try-uiip-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 17), 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 25), it is performed updating C_{try} . In addition, all new literals at level i are added to L_i . These are the literals at level i in the reason clause.

On the other hand, if the resolution step would introduce new decision levels (line 20) then there are two options. The first option we call *pure-allUIP*. With *pure-allUIP* we abort our attempt to UIP this level and return the clause with level i unchanged. In the second option, called *min-allUIP*, we continue without performing the resolution, *keeping* the current literal p in C_{try} . *min-allUIP* then continues 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.

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 *stable-allUIP***Require:** C_1 is minimized 1-UIP clause**Require:** $t_{gap} \geq 0$ is global parameter, n_{tries} and n_{succ} are globals used to configure t_{gap}

```

1: stable-allUIP( $C_1, \mathcal{T}, type$ )
2:   if ( $|C_1| - \text{LBD}(C_1) < t_{gap}$ ) return  $C_1$ 
3:    $C_i \leftarrow C_1$ 
4:    $decLvl s \leftarrow$  decision levels in  $C_i$  in descending order ▷ These never change
5:   for ( $i = 1; i < |decLvl s|; i++$ ) ▷ skip the deepest level  $decLvl s[0]$ 
6:      $C_i \leftarrow \text{try-uip-level}(C_i, decLvl s[i], type)$  ▷ Try to reduce this level to UIP
7:     if  $|\{\ell \mid \ell \in C_i \wedge decLvl(\ell) \geq decLvl s[i]\}| + (|decLvl s| - (i + 1)) \geq |C_1|$ 
8:       return  $C_1$ . ▷ can't generate smaller clause
9:   if ( $type = \text{pure-allUIP}$ )  $C_i \leftarrow \text{minimize}(C_i)$ 
10:   $n_{tries}++$ 
11:  if ( $|C_i| < |C_1|$ )  $n_{succ}++$ , return  $C_i$ 
12:  ▷ Use next line instead for allUIP-Active
13:  if ( $|C_i| < |C_1| \wedge (\text{AvgVarAct}(C_i) > \text{AvgVarAct}(C_1))$ )  $n_{succ}++$ , return  $C_i$ 
14:  return  $C_1$ 

15: try-uip-level( $C_i, i, type$ ) ▷ Do not add new decision levels
16:   $C_{try} = C_i$ 
17:   $L_i = \{\ell \mid \ell \in C_{try} \wedge decLvl(\ell) = i\}$ 
18:  while  $|L_i| > 1$ 
19:     $p \leftarrow$  remove lit with the highest trail index from  $L_i$ 
20:    if ( $\exists q \in \text{reason}(\neg p). decLvl(q) \notin decLvl s$ ) ▷ Would add new decision levels
21:      if ( $type = \text{pure-allUIP}$ )
22:        return  $C_i$  ▷ Abort, can't UIP this level
23:      else if ( $type = \text{min-allUIP}$ )
24:        continue ▷ Don't try to resolve away  $p$ 
25:      else
26:         $C_{try} \leftarrow C_{try} \bowtie \text{reason}(\neg p)$ 
27:         $L_i = L_i \cup \{\ell \mid \ell \in \text{reason}(\neg p) \wedge \ell \neq \neg p \wedge decLvl(\ell) = i\}$ 
28:  return  $C_{try}$ 

```

Hence, *min-allUIP* can leave multiple literals at level i —all of those with reasons containing new levels along with one other.³ Observe that the number of literals at level i can not be increased after processing it with *pure-allUIP*. *min-allUIP* can, however, potentially increase the number of literals at a level. In resolving away a literal l at level i more literals might be introduced into level i , and some of these might not be removable by *min-allUIP* if their reasons contain new levels. However, both *pure-allUIP* and *min-allUIP* can increase the number of literals at levels less than i as new literals can be introduced into those levels when the literals at level i are resolved away. These added literals at the lower levels might not be removable from the clause, and thus both methods might yield a longer clause than the inputted 1-UIP clause.

³ Since the sole remaining literal $u \in L_i$ is at a lower trail index than all of the other 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-allUIP* we can once again apply minimization (line 9). Recursive clause minimization [19] would be useless for the *min-allUIP* clause as all but one literal of each level introduces a new level and thus cannot be recursively removed.⁴

Finally, the Algorithm returns the shorter of the initial 1-UIP clause and the newly computed clause (line 11). 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 $decLvls[0] \dots decLvls[i]$ the literals at those levels will not 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. As observed above, *pure-allUIP* cannot trigger this early termination condition (it cannot increase the size of any processed level), but *min-allUIP* can.

t_{gap} : *stable-allUIP* can produce significantly smaller clause. However, when it does not yield a smaller clause, the cost of the additional resolution steps can hurt the solver's performance. Since resolution cannot reduce a clause's LBD, The maximum size reduction obtainable from *stable-allUIP* is the difference between the 1-UIP clause's size and its LBD: $Gap(C_1) = |C_1| - LBD(C_1)$. When $Gap(C_1)$ is small, applying *stable-allUIP* is unlikely to be cost effective. Our approach is to dynamically set a threshold on $Gap(C_1)$, t_{gap} , such that when $Gap(C_1) < t_{gap}$ we do not attempt to reduce the clause (line 2). Initially, $t_{gap} = 0$, and we count the number of times *stable-allUIP* is attempted (n_{tries}) and the number of times it successfully yields a shorter clause (n_{succ}) (line 10 and 11). On every restart if the success rate is less than 80%, we decrease t_{gap} by one (not allowing it to become negative), and if it is greater than 80% we increase t_{gap} by one.

Example 1. Consider the trail $\mathcal{T} = \dots, \ell_1, a_2, b_2, c_2, d_2, \dots, e_5, f_5, g_5, h_6, i_6, j_6, k_6, \dots, m_{10}, \dots$ 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

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

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. *stable-allUIP* 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.

stable-allUIP 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-allUIP* thus leaves level 5 unchanged. *min-allUIP*, on the other hand, skips the

⁴ Other more powerful minimization techniques could still be applied.

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-allUIP* does not do any further resolutions at this level. Hence, *pure-allUIP* yields C^* unchanged, while *min-allUIP* 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, *stable-allUIP* 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-allUIP* will leave level 2 unchanged and *min-allUIP* will leave b_2 unresolved. The final clauses produced by *pure-allUIP* 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-allUIP* 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. \square

3.1 Variants of *stable-allUIP*

We also developed and experimented with a few variants of the *stable-allUIP* algorithm which describe below.

allUIP-Active: Clauses with Active Variables. *stable-allUIP* learning might introduce literals with low variable activity into C_i . Low activity variables are variables that have had low recent participation in clause learning. Hence, clauses with variables of low activity might not be as currently useful to the solver. Our variant *allUIP-Active*, shown as the optional line 13 in Algorithm 1, computes the average variable activity of the newly produced all-UIP clause C_i and the original 1-UIP clause C_1 . The new clause C_i will be returned only if it is both smaller and has higher average variable activity than the original 1-UIP clause. There are, of course, generalizations of this approach where one has a weighted tradeoff between these factors that allows preferring the new clause when it has large gains in one metric even though it has small losses in the other. We did not, however, experiment with such generalization.

Adjust Variable Activity. An alternative to filtering clauses with low average variable (*allUIP-Active*) is to alter the way variable activities are updated to account for our new clause learning method. The popular branching heuristics VSIDS [12] and LBR [6] bump the variable activity for all literals appearing in the learnt clause C_L and all literals resolved away during the conflict analysis that yielded C_L from the initially detected conflict C_I (all literals on the conflict side).

We did not apply this approach to the *stable-allUIP* clause, as we did not want to bump the activity of the literals at above the deepest decision level that *stable-allUIP* resolves away. Intuitively, these literals did not directly contribute to generating the conflict. Instead, we tried two modifications to the variable activity bumping schemes.

Let C_1 be the 1-UIP learnt clause and C_i be the *stable-allUIP* learnt clause. First, we kept all of the variable activity bumps normally done by 1-UIP learning.⁵ Then, when the *stable-allUIP* scheme was successful, i.e., C_i was to be used as the new learnt clause, we perform further updates to the variable activities. In the *allUIP-Inclusive*

⁵ So extra techniques used by the underlying solver, like reason side rate and locality [6], were kept intact.

approach all variables appearing in C_i that are not in C_1 have their activities bumped. Intuitively, since the clause C_i is being added to the clause database we want to increase the activity of all of its variables. On the other hand, in the *allUIP-Exclusive* approach in addition to bumping the activity of the new variables in C_i we also remove the activity bumps of those variables in C_1 that are no longer in C_i .

In sum, the two modified variable activity update schemes we experimented with were

$$\text{allUIP-Inclusive} \equiv \forall l \in C_i - C_1. \text{bumpActivity}(l)$$

$$\text{allUIP-Exclusive} \equiv \forall l \in C_i - C_1. \text{bumpActivity}(l) \wedge \forall l \in C_1 - C_i. \text{unbumpActivity}(l)$$

Chronological Backtracking. We tested our new clause learning schemes on solvers that utilized Chronological Backtracking [13,11]. When chronological backtracking is used the literals on the trail might no longer be sorted by decision level. So resolving literals in the conflict by highest trail index first no longer works. However, we can define a new ordering on the literals to replace the trail index ordering. Let l_1 and l_2 be two literals on the trail \mathcal{T} . We say that $l_1 >_{\text{chron}} l_2$ if $\text{decLvl}(l_1) > \text{decLvl}(l_2) \vee (\text{decLvl}(l_1) = \text{decLvl}(l_2) \wedge \iota(l_1) > \iota(l_2))$. That is, literals with higher decision level come first, and if that is equal then the literal with higher trail index comes first.

Exploiting the analysis of [11] it can be observed that all clause learning schemes continue to work as long literals are resolved away from the initial conflict in decreasing $>_{\text{chron}}$ order. In our implementation we used a heap (priority queue) to achieve this ordering of the literal resolutions in order to add our new schemes to those solvers using chronological backtracking.

4 Implementation and Experiments

We implemented *stable-allUIP* learning schemes on *MapleCOMSPS_LRB* [7], the winner of SAT Race 2015 application track. We then evaluated these schemes and compare against 1-UIP baseline on the full set of benchmarks from SAT RACE 2019 main track which contains 400 instances. We ran our experiments on 2.70 GHz XeonE5-2680 CPUs, allowing 5000 seconds per instance and a maximum of 12 GB memory.

Solver	# solved (SAT, UNSAT)	PAR-2	Avg clause Size
1-UIP	221 (132, 89)	5018.89	62.6
<i>pure-allUIP</i>	228 (135, 93) +7	4867.37	49.88
<i>min-allUIP</i>	226 (135, 91) +5	4890.67	45.2
<i>allUIP-Active</i>	226 (135, 91) +5	4866.94	47.7
<i>allUIP-Inclusive</i>	225 (138 , 87) +4	4958.49	52..12
<i>allUIP-Exclusive</i>	223 (134, 89) +2	5015.23	43.2

Fig. 2. Benchmark results of *MapleCOMSPS_LRB* with 1-UIP, *pure-allUIP*, *min-allUIP*, *allUIP-Active*, *allUIP-Inclusive*, and *allUIP-Exclusive* on SAT2019 race main track.

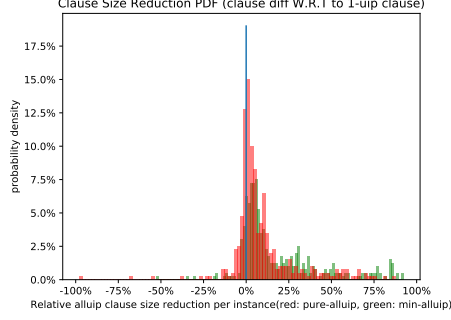


Fig. 3. Relative clause size reduction distribution. X axis indicates the relative size of difference between i-UIP and 1-UIP clauses (calculated as $\frac{|C_1| - |C_i|}{|C_1|}$) for each instance, and Y axis shows the probability density.

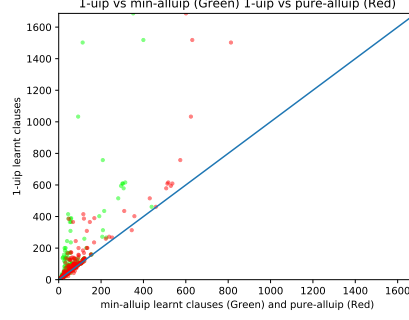


Fig. 4. Average clause size comparison plot. Each point in the plot represents an instance. X and Y axis shows the clause length from *stable-allUIP* and 1-UIP, respectively. Each green (red) dot represents an compared instance between *MapleCOMSPS_LRB* and *Maple-min-allUIP* (*pure-allUIP*).

Fig. 2 shows each learning scheme’s solved instances count, PAR-2 score, and average clause size⁶. *stable-allUIP* learning schemes improved solved instances, and PAR-2 scores over 1-UIP. More specifically, *pure-allUIP* solved the most instances (+7 over 1-UIP) and the most UNSAT instances (+4); *allUIP-Inclusive* solved the most SAT instances (+6); and *allUIP-Active* yields the best PAR-2 score (-151 than 1-UIP). Besides the performance gain, *stable-allUIP* schemes yield significantly smaller average clause size.

Clause Reduction with *stable-allUIP* To precisely measure *stable-allUIP* clause reduction power, we compare each instance’s learnt clause size from *min-allUIP* and *pure-allUIP* against 1-UIP. Fig. 3 shows the probability density distribution (PDF) of the relative clause size of *stable-allUIP* learning schemes (*min-allUIP* in green and *pure-allUIP* in red) for each instance. *min-allUIP* (*pure-allUIP* resp.) produces shorter clauses for 88.5% (77.7%) of instances, and the average relative reduction ratio over 1-UIP is 18.6% (16.7%). Fig. 4 compares the absolute clause size of *min-allUIP*, *pure-allUIP* and 1-UIP per instance. Both *stable-allUIP* schemes generally yield smaller clauses, and the size reduction is more significant for instances with larger 1-UIP clause.

Horizontally, The results in Fig. 2, 3 and 4 indicate *min-allUIP* often achieves higher clause reduction than *pure-allUIP*. We also observed that *min-allUIP* attempted algorithm 1 more frequently than *pure-allUIP* (28.8% vs 16.1%), and is more likely to succeed (59.3% vs 43.4%). This observation agrees with our experiment results.

⁶ The average clause size is measured by dividing learnt literals by learnt clauses.

Proof Reduction with *stable-allUIP* A learning scheme yields smaller clauses (lemmas) may construct smaller causal proofs. For UNSAT instances, we additionally compare the size of the optimized DRAT proof from *pure-allUIP*, *min-allUIP* and 1-UIP schemes. We used DART-trim [21] with 10000 second timeout to check and optimize every DRAT proof once ⁷.

The average optimized DRAT proof from *min-allUIP* and *pure-allUIP* are 413.2MB and 487.2MB, respectively. Both sizes are significantly smaller than the average optimized proof size from 1-UIP, 613.9MB. The average relative proof size reduction per instance of *min-allUIP* and *pure-allUIP* are 17.18% and 6.9% compare to 1-UIP, which roughly correlate to the observation from relative clause reduction plot in Fig 3. Moreover, comparing the absolute proof size between *min-allUIP* and 1-UIP yields a similar trend as clause size comparison in Fig 4.

***stable-allUIP* on Modern SAT solvers** To validate *stable-allUIP* as practical learning schemes on modern SAT solvers, we re-implement *stable-allUIP* on the winners of 2017, 2018 and 2019 SAT Race[9,15,20] and *expMaple_CM_GCBumpOnlyLRB*[10] (*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 random walk simulation branching heuristic is different from local branching heuristic (VSIDS and LRB) that we have considered, hence it is in the blind spot of *stable-allUIP*. we compare the base 1-UIP learning scheme against *pure-allUIP*, *min-allUIP* and the top two *stable-allUIP* variants, *allUIP-Active* and *allUIP-Inclusive*, on the SAT Race 2019 main track benchmark. We report solved instances, PAR-2 score and the average clause size.

Table 5 shows the benchmark result of *stable-allUIP* configurations on modern solvers. Overall, we observed similar performance gain on all modern solvers as we have seen on *MapleCOMSPS_LRB* in Fig. 2. More specifically, almost all configurations improved on solved instance (+4.5 instances in average) and PAR-2 score (-83.6 in average). The average clause size reduction is consistent across all solvers. Each configuration also exhibits different strength: *pure-allUIP* solved the most instances with the best PAR-2 score on two solvers, *min-allUIP* yields small clauses, *allUIP-Inclusive* solved the most SAT instances, and *allUIP-Active* has stable performance.

On the SAT 2017 race winner *MapleLCMDist*, all four configurations of *stable-allUIP* outperformed 1-UIP learning by 12, 8, 5, and 2 instances, respectively. *pure-allUIP* solved more UNSAT and SAT instances while other configurations improved on solving SAT instances. The clause size reduction of *stable-allUIP* is more significant on *MapleLCMDist* compare to *MapleCOMSPS_LRB*.

The SAT 2018 race winner *MapleCB* uses chronological backtracking (CB) for long distance backtracks. Therefore, we applied *stable-allUIP*-CB extension for all *stable-allUIP* configurations. Three out of four *stable-allUIP* configurations outperformed 1-UIP by 5, 4 and 4 instances, respectively.

On the SAT 2019 race winner *MapleCB-DL*, three out of four *stable-allUIP* configurations solved instances 6, 5 and 5 more instances than 1-UIP, respectively. The

⁷ Applying DART-trim multiple times can further reduce the proof size until fix-point, but the full optimization is time consuming and impractical for our experiments.

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

Fig. 5. Benchmark results of 1-UIP, *pure-allUIP*, *min-allUIP*, *allUIP-Active* and *allUIP-Inclusive* on SAT2019 race main track.

solver uses duplicate learning (DL) to prioritize clauses that are learned multiple times. We observed that clauses learnt from *stable-allUIP* configurations are less likely to be duplicated. As an example, *min-allUIP* in average, added 12% less duplicated clauses into the core clause database than 1-UIP. This observation is surprising, and the cause is unclear.

On *expMaple*, three out of four *stable-allUIP* configurations solved 4, 7 and 8 more instances than 1-UIP learning. We noticed that *allUIP-Active* and *allUIP-Inclusive* show significant better performance than *min-allUIP* and *pure-allUIP* on this solver. The random walk simulation branching heuristic however didn't impact the performance of *stable-allUIP* schemes significantly.

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. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9710, pp. 123–140. Springer (2016), https://doi.org/10.1007/978-3-319-40970-2_9
7. Liang, J.H., Oh, C., Ganesh, V., Czarnecki, K., Poupart, P.: Maple-comsps, maplecomsps lrb, maplecomsps chb. SAT Competition p. 52
8. 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>
9. Luo, M., Li, C.M., Xiao, F., Manyà, F., Lü, Z.: An effective learnt clause minimization approach for cdcl sat solvers. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence. pp. 703–711 (2017)

10. Md Solimul, M., Jia-Huai: Four cdcl sat solvers based on exploration and glue variable bumping. *Proceedings of SAT competition* p. 17 (2019)
11. Möhle, S., Biere, A.: Backing backtracking. In: Janota, M., Lynce, I. (eds.) *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science*, vol. 11628, pp. 250–266. Springer (2019), https://doi.org/10.1007/978-3-030-24258-9_18
12. 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>
13. Nadel, A., Ryvchin, V.: Chronological backtracking. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings. Lecture Notes in Computer Science*, vol. 10929, pp. 111–121. Springer (2018), https://doi.org/10.1007/978-3-319-94144-8_7
14. 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>
15. Ryvchin, V., Nadel, A.: Maple_lcm_dist chronobt: Featuring chronological backtracking. *Proceedings of SAT competition* p. 29 (2018)
16. 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>
17. 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>
18. 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>
19. 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
20. Stepan, Oleg, V., Alexander: Maplelcmdistchronobt-dl, duplicate learnts heuristic-aided solvers at the sat race 2019. *Proceedings of SAT competition* p. 24 (2019)
21. Wetzler, N., Heule, M.J., Hunt, W.A.: Drat-trim: Efficient checking and trimming using expressive clausal proofs. In: *International Conference on Theory and Applications of Satisfiability Testing*, pp. 422–429. Springer (2014)
22. Wieringa, S., Heljanko, K.: Concurrent clause strengthening. In: Jarvisalo, 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
23. 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, Novem-*

ber 4-8, 2001. pp. 279–285. IEEE Computer Society (2001), <https://doi.org/10.1109/ICCAD.2001.968634>