Clause Size Reduction with all-UIP Learning

Nick Feng(✉) and Fahiem Bacchus(✉)

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

Abstract. Almost all CDCL SAT solvers use the 1-UIP clause learning scheme for learning new clauses from conflicts, and our current understanding of SAT solving provides good reasons for using that scheme. In particular, the 1-UIP scheme yields asserting clauses, and these asserting clauses have minimum LBD among all possible asserting clauses. As a result of these advantages, other clause learning schemes, like i-UIP and all-UIP, that were proposed in early work are not used in modern solvers. In this paper, we propose a new technique for exploiting the all-UIP clause learning scheme. Our technique is to employ all-UIP learning under the constraint that the learnt clause's LBD does not increase (over the minimum established by the 1-UIP clause). Our method can learn clauses that are significantly smaller than the 1-UIP clause while preserving the minimum LBD. Unlike previous clause minimization methods, our technique is not limited to learning a sub-clause of the 1-UIP clause. We show empirically that our method can improve the performance of state of the art solvers.

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 [6]. In early SAT research a number of different clause learning techniques were proposed [5,19,20,25]. 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 [25] employed in Chaff [13] (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 [21], as their primary method for learning new clauses.

However, other clause learning schemes can be used in SAT solvers without changes to the 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 these previously proposed schemes with a focus on the schemes described in [25]. Improved understanding

[1] The idea of UIP clauses was first mentioned in [19], and 1-UIP clauses along with other UIP clauses were learnt and used in the earlier GRASP SAT solver.

L. Pulina and M. Seidl (Eds.): SAT 2020, LNCS 12178, pp. 28–45, 2020.
https://doi.org/10.1007/978-3-030-51825-7_3

of SAT solvers, obtained from the last decade of research, allows us to see that in their original form these alternative clause learning schemes suffer significant disadvantages over 1-UIP clause learning.

One of the previously proposed schemes was the all-UIP scheme [25]. In this paper we propose a new way to exploit the main ideas of this scheme that avoids its main disadvantage which is that it can learn clauses with higher LBD scores. 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., [10,21,24]. Our method is orthogonal to clause minimization. In particular, our approach can learn a clause that is not a subset of the 1-UIP clause but which still serves all of the same purposes as the 1-UIP clause. Clause minimization techniques can be applied on top of our method to remove redundant literals.

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.

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 $C_1 \bowtie C_2$ 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 x and its negation $\neg x$) 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 [18].

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 $\mathit{decLvl}(\ell)$. Let k be the number of decision literals appearing before ℓ on the trail. When ℓ is a unit propagated literal $\mathit{decLvl}(\ell) = k$, and when ℓ is a decision literal $\mathit{decLvl}(\ell) = k + 1$. For example, $\mathit{decLvl}(d) = 1$ for the first decision literal $d \in \mathcal{T}$, and $\mathit{decLvl}(\ell) = 0$ for all literals ℓ appearing before d on the trail. 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 $\mathit{decLvl}(d_i) = i$ we call this subsequence of $\mathcal{T}$ the *i-th decision level.*

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

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. We will try to make as few assumptions about how $\mathcal{T}$ is managed by the SAT solver as possible. One assumption we will make is that $\mathcal{T}$ *remains intact during clause learning* and is only changed after the new clause is learnt (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 **unit 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 $\mathit{decLvl}(x) \leq \max\{\mathit{decLvl}(y) | y \in C_u \wedge y \neq x\}$. 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 unit 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 $\mathit{reason}(\ell)$ 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 $\mathrm{var}(\ell)$ 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, ..., 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. W.l.o.g 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, ..., $C_m = C_L$ obtained by resolving on the sequence of variables v_1, ..., v_m whose corresponding literals on $\mathcal{T}$ are l_1, ..., 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 $C'_L = \bigcup_{i=1}^{m} \mathit{reason}(l_i) - \{l_1, \neg l_1, \ldots, l_m, \neg l_m\}$. Since U resolves on the same variables (in a different order) 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 $\iota(l_{i+1}) > \iota(l_i)$. □

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 [21] 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. Observation 1 shows that this is correct.

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 [20] 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., [5,19,20,25].

(a) All Decision Clause

```
all-decision($C_I$)
  $C \leftarrow C_I$
  while $\{l \mid l \in C \wedge reason(l) \neq \varnothing\} \neq \emptyset$
      $\ell \leftarrow$ literal with highest trail index in
          $\{l \mid l \in C \wedge reason(l) \neq \varnothing\}$
      $C \leftarrow C \bowtie reason(\neg\ell)$
  return $C$
```

(b) Make level i contain a single literal

```
UIP-level($C$, $i$)
  while $\left|\left\{\ell \,\middle|\, \begin{array}{l} \ell \in C \\ \wedge\, reason(l) \neq \varnothing \\ \wedge\, decLvl(l) = i \end{array}\right\}\right| > 1$
      $l \leftarrow$ literal with highest trail index in
          $\left\{\ell \,\middle|\, \begin{array}{l} \ell \in C \wedge reason(l) \neq \varnothing \\ \wedge\, decLvl(l) = i \end{array}\right\}$
      $C \leftarrow C \bowtie reason(\neg l)$
  return $C$
```

(c) First UIP Clause

```
1-UIP($C_I$)
  $i \leftarrow \max\{decLvl(l) \mid l \in C_I\}$
  return UIP-level($C_I$, i)
```

(d) all-UIP Clause

```
all-UIP($C_I$, $i$)
  $i = |\{decLvl(l) \mid l \in \mathcal{T}\}|$
  ▷ $i$ is large enough to ensure all levels are UIP
  return $i$-UIP(C, i)
```

(e) i-UIP Clause

```
$i$-UIP($C_I$, $i$)
  $C \leftarrow C_I$
  $d \leftarrow \max\{decLvl(l) \mid l \in C\}$
  for ($j \leftarrow 1$; $j \leq i$; $j \leftarrow j + 1$)
      if ($d = \varnothing$): break
      $C \leftarrow$ UIP-level(C, d)
      $d \leftarrow \max\left\{decLvl(l) \,\middle|\, \begin{array}{l} l \in C \\ \wedge\, decLvl(l) < d \end{array}\right\}$
  return $C$
```

Maximum of an empty set is $\varnothing$

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.

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 [20] 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, and in some contexts the term all-UIP is used to refer to all the unique implication points that exist in a particular decision level [17] rather than the all-UIP clause learning scheme as is used here. The algorithms given in Fig. 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** [15] is a conflict clause C_L that has exactly one literal ℓ at its deepest level, i.e., $\forall x \in C_L . \mathit{decLvl}(x) \leq \mathit{decLvl}(\ell) \wedge (\mathit{decLvl}(x) = \mathit{decLvl}(\ell) \rightarrow x = \ell)$. All of the clause learning schemes in Fig. 1 produced learnt clauses that are asserting.

The main advantage of asserting clauses is that they are 1-Empowering [15], 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 unit 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.

Example 1. Suppose that a is a unit propagated literal and d is a decision literal with $\mathit{decLvl}(d) > \mathit{decLvl}(a)$. Let the sequence of clauses watched by $\neg d$ be $(\neg d, x, \neg a)$, $(\neg d, y, \neg x, \neg a)$, $(\neg d, \neg y, \neg x, \neg a)$, $(\neg d, \neg x, \neg a)$. When d is unit propagated the clauses on $\neg d$'s watch list will be checked in this order.

Hence, unit propagation of d will extend the trail by first adding the unit propagated literal x (with $\mathit{reason}(x) = (x, \neg d, \neg a)$) and then the unit propagated literal y (with $\mathit{reason}(y) = (y, \neg x, \neg a, \neg d)$). Now the third clause on $\neg d$'s watch list, $(\neg d, \neg y, \neg x, \neg a)$ is detected to be a conflict.

Clause learning can now be initiated from conflict $C_I = (\neg d, \neg y, \neg x, \neg a)$. This clause has 3 literals at level $\mathit{decLvl}(d) = 10$. If we stop clause learning before reaching an asserting clause, then it is possible to simply resolve C_I with $\mathit{reason}(y)$ to obtain the learnt clause $C_L = (\neg d, \neg x, \neg a)$. However, this non-asserting learnt clause is a duplicate of the fourth clause on $\neg d$'s watch list which is already in the formula.[2] This issue can arise whenever C_L contains two or more literals at the deepest level (i.e., whenever C_L is not asserting). In such cases C_L might be a clause already in the formula with its two watches not yet fully unit propagated (and thus C_L is not detected by the SAT solver to be a conflict) since propagation is typically stopped as soon as a conflict is detected.

The LBD of the learnt clause C_L is the number of different decision levels in it: $\mathrm{LBD}(C_L) = \left|\left\{ \mathit{decLvl}(l) \,|\, l \in C_L \right\}\right|$ [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 $\mathit{reason}(l)$, then $\mathrm{LBD}(C') \geq \mathrm{LBD}(C)$ since $\mathit{reason}(l)$ must contain at least one other literal

[2] In this example, the fourth clause on $\neg d$'s watch list subsumes the third clause. But it is not difficult to construct more elaborate examples where there are no subsumed clauses and we still obtain learnt clauses that are duplicates of clauses already in the formula.

with the same decision level as l and might contain literals with decision levels not in C. That is, the each trail resolution step might increase the LBD of the learnt clause and can never decrease the LBD. 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 addition decision levels into the final clause. Hence, these schemes learn clauses with LBD at least as large as the 1-UIP clauses.

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 solvers almost exclusively use 1-UIP clause learning.[3]

3 Using all-UIP Clause Learning

Although learning clauses with low LBD has been shown empirically to be more 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 more 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 Fig. 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 [21] to the 1-UIP clause. That is, our algorithm *stable-alluip* (Algorithm 1), starts with the clause that most SAT solvers learn from a conflict, a minimized 1-UIP clause.

Algorithm 1 tries to compute a clause shorter than the inputted 1-UIP clause C_1. If a clause shorter than C_1 cannot be computed the routine returns C_1 unchanged. Line 2 uses the parameter t_{gap} to predict if Algorithm 1 will be successful in producing a shorter clause. This predication is described below. If the prediction is negative C_1 is immediately returned and Algorithm 1 is not attempted. Otherwise, a copy of C_1 is made in C_i and n_{tries}, which counts the number of times Algorithm 1 is attempted, is incremented.

[3] Knuth in his sat13 CDCL solver [7] 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: $config$ a set of configuration parameters to give different versions $stable\text{-}alluip$.
Require: $t_{gap} \geq 0$ is a global parameter, $n_{tries}$ and $n_{succ}$ are used to dynamically adjust $t_{gap}$
 1: $stable\text{-}alluip(C_1, \mathcal{T})$
 2:     if $(|C_1| - \mathrm{LBD}(C_1) < t_{gap})$ return $C_1$
 3:     $n_{tries}{+}{+}$
 4:     $C_i \leftarrow C_1$
 5:     $decLvls \leftarrow$ decision levels in $C_1$ in descending order        ▷ These never change
 6:     for $(i = 1;\ i < |decLvls|;\ i{+}{+})$                  ▷ skip the deepest level $decLvls[0]$
 7:         $C_i \leftarrow try\text{-}uip\text{-}level\ (C_i,\ decLvls[i])$          ▷ Try to reduce this level to UIP
 8:         if $\left|\{\ell \,|\, \ell \in C_i \wedge decLvl(\ell) \geq decLvls[i]\}\right| + (|decLvls| - (i+1)) \geq |C_1|$
 9:             return $C_1$                                  ▷ can't generate smaller clause
10:     if $pure\text{-}alluip \in config$
11:         $C_i \leftarrow$ minimize$(C_i)$
12:     if $\big(|C_i| < |C_1| \wedge alluip\text{-}active \in config \rightarrow (\mathrm{AvgVarAct}(C_i) > \mathrm{AvgVarAct}(C_1))$
13:         $n_{succ}{+}{+}$, return $C_i$                    ▷ $C_i$ is smaller than the input clause
14:     else
15:         return $C_1$

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

Then the decision levels of C_1 are computed and stored in *decLvls* in order from largest to lowest. The for loop of lines 6–9 is then executed for each decision level *decLvls*$[i]$. In the loop the subroutine *try-uip-level* tries to reduce the set of literals at *decLvls*$[i]$ down to a single UIP literal using a sequence of trail resolutions. Since C_1 is a 1-UIP clause *decLvls*$[0]$ (the deepest level) already contains only one literal, so we can start at $i = 1$.

After the call to *try-uip-level* a check (line 8) is made to see if we can abort further processing. At this point the algorithm has finished processing levels *decLvls*$[0]$–*decLvls*$[i]$ so the literals at those levels will not change. Furthermore, we know that the best that can be done from this point on is to reduce the remaining $|decLvls| - (i + 1)$ levels down to a single literal each. Hence, adding these two numbers gives a lower bound on the size of the final computed clause.

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.

After processing all decision levels, if *try-uip-level* is using the *pure-alluip* configuration, additional reduction in the clause size might be achieved by an another round of clause minimization (line 11). Finally, if the newly computed clause C_i is smaller that the input clause C_1 it is returned. Otherwise the original clause C_1 is returned. Additionally, if the configuration *alluip-active*, described in Sect. 3.1, is being used, then we also require that the average activity level of the new clause C_i be larger than C_1 before we can return the new clause C_i.

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 18), 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 26), it is performed updating C_{try}. In addition, all new literals added to C_{try} at level i are added to L_i.

On the other hand, if the resolution step would introduce new decision levels (line 21) 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. Hence, *min-alluip* can leave multiple literals at level i—all of those with reasons containing new levels along with one other.[4] 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 level i. 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 that 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 can yield a longer clause than the input 1-UIP clause.

After trying to UIP each level the clause C_i is obtained. If we were using *pure-alluip* we can once again apply recursive clause minimization (line 11) [21], but this would be useless when using *min-alluip* as all but one literal of each level introduces a new level and thus cannot be recursively removed.[5]

t_{gap}: *stable-alluip* can produce significantly smaller clauses. 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: $\text{gap}(C_1) = |C_1| - \text{LBD}(C_1)$. When $\text{gap}(C_1)$

[4] Since the sole remaining literal $u \in L_i$ is at a lower trail index than all of the other literals there is no point in trying to resolve away u—either it will be the decision literal for level i having no reason, or its reason will contain at least one other literal at level i.

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

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 3 and 13). On every restart if the success rate since the last restart is greater than 80% (less than 80%), we decrease (increase) t_{gap} by one not allowing it to become negative.

Example 2. 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.

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

Let the clauses C_x, show above, denote the reason clause for literal x_i. 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 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* yields $C^*_{min} = (\neg m_{10}, \neg h_6, \neg f_5, \neg e_5, \neg d_2, \neg c_2, \neg a_2)$.

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. □

3.1 Variants of *stable-alluip*

We also developed and experimented with a few variants of the *stable-alluip* algorithm which we 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* (line 12) 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 trade-off 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 generalizations.

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 [13] and LBR [8] 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 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.[6] 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* approach all variables 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 (1) ***alluip-inclusive*** $\equiv \forall l \in C_i - C_1.\, \mathit{bumpActivity}(l)$ and (2) ***alluip-exclusive*** $\equiv \forall l \in C_i - C_1.\, \mathit{bumpActivity}(l) \wedge \big(\forall l \in C_1 - C_i.\, \mathit{unbumpActivity}(l)\big)$.

Chronological Backtracking. We tested our new clause learning schemes on solvers that utilized Chronological Backtracking [12,14]. 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 $l_1 >_{chron} l_2$ if $\mathit{decLvl}(l_1) > \mathit{decLvl}(l_2) \vee (\mathit{decLvl}(l_1) = \mathit{decLvl}(l_2) \wedge \iota(l_1) > \iota(l_2))$. That is, literals with higher decision level come first, and if that is equal then the literal with higher trail index comes first.

[6] So extra techniques used by the underlying solver, like reason side rate and locality [8], were kept intact.

Exploiting the analysis of [12], it can be observed that all clause learning schemes continue to work as long as literals are resolved away from the initial conflict in decreasing $>_{chron}$ 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* [9], the winner of SAT Race 2016 application track. We then evaluated these schemes and compare against the 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
pure-alluip	**228** (135, **93**) **+7**	4867.37	49.88
min-alluip	226 (135, 91) +5	4890.67	45.2
alluip-active	226 (135, 91) +5	**4866.94**	47.7
alluip-inclusive	225 (**138**, 87) +4	4958.49	52.12
alluip-exclusive	223 (134, 89) +2	5015.23	**43.2**

Fig. 2. Results of *MapleCOMSPS-LRB* with 1-UIP, *pure-alluip*, *min-alluip*, *alluip-active*, *alluip-inclusive*, and *alluip-exclusive* on SAT2019 race main track.

Figure 2 shows each learning scheme's solved instances count, PAR-2 score, and average learnt clause size. We found that the *stable-alluip* learning schemes improved solved instances, PAR-2 scores, and learnt clause size 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). In all cases the *stable-alluip* schemes learnt significantly smaller clauses on average.

Clause Reduction with *stable-alluip*. To precisely measure *stable-alluip*'s clause reduction power, we compare each instance's learnt clause size from *min-alluip* and *pure-alluip* against 1-UIP. Figure 3 shows the probability density distribution (PDF) of the relative clause size of the *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 18.5% (9.6%). Figure 4 compares the average learnt 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 clauses.

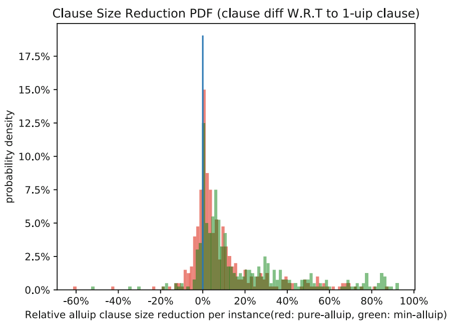

Fig. 3. Relative clause size reduction distribution. The X axis indicates the relative size of difference between all-UIP and 1-UIP clauses (calculated as $\frac{|C_1| - |C_i|}{|C_1|}$) for each instance, and the Y axis shows the probability density. (Color figure online)

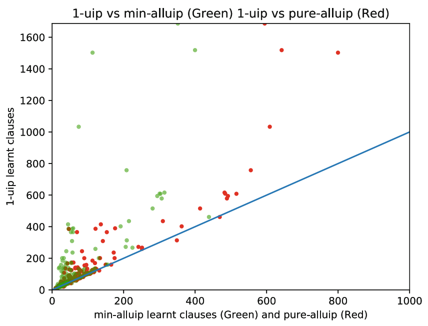

Fig. 4. Average clause size comparison plot. Each point in the plot represents an instance. The X and Y axes shows the clause length from *stable-alluip* and 1-UIP, respectively. Each green (red) dot represents an compared instance between *MapleCOMSPS-LRB* and Maple-*min-alluip* (*pure-alluip*). (Color figure online)

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.

Reduced Proof Sizes with *stable-alluip*. A learning scheme that yields smaller clauses (lemmas) might also construct smaller causal proofs. For 88 UNSAT instances solved mutually by *pure-alluip*, *min-alluip* and 1-UIP schemes, we additionally compared the size of the optimized DRAT proof from the three learning schemes. We used the DRAT-trim tool [23] with a 30000 second timeout to check and optimize every DRAT proof once[7].

The average optimized DRAT proof from *min-alluip* and *pure-alluip* are 556.6MB and 698.5MB, respectively. Both sizes are significantly smaller than the average optimized proof size from 1-UIP, 824.9MB. The average proof size reduction per instance for *min-alluip* and *pure-alluip* is 16.5% and 3.6% against 1-UIP, which roughly correlate with our clause size observation in Fig. 3.

***stable-alluip* in Modern SAT Solvers.** To validate *stable-alluip* in modern SAT solvers, we implemented *stable-alluip* in the winners of 2017, 2018 and 2019 SAT Race [10,16,22] and in the *expMaple-CM-GCBumpOnlyLRB* [11] (*expMaple*) and *CaDiCaL* [4] solvers. *expMaple* is a top ten solver from 2019 SAT race which uses random walk simulation to help branching. We chose *expMaple* because the random walk simulation branching heuristic is different from local branching heuristic (VSIDS and LRB) that we have considered in

[7] Applying DRAT-trim multiple times can further reduce the proof size until a fixpoint. However, the full optimization is too time consuming for our experiments.

Solver	#solved (SAT, UNSAT) Δ	PAR-2	avg. clause Size
SAT 2017 Winner *MapleLCMDist*	232 (135, 97)	4755.96	61.9
MapleLCMDist-all-pure	**244 (146, 98)** +12	**4504.18**	43.76
MapleLCMDist-all-min	240 (144, 96) +8	4601.25	**36.97**
MapleLCMDist-all-act	237 (140, 97) +5	4678.434	43.62
MapleLCMDist-all-inclusive	234 (137, 97) +2	4718.03	37.96
SAT 2018 Winner *MapleCB*	236 (138, 98)	4671.81	61.69
MapleCB-all-pure	**241** (**142, 99**) +5	**4598.18**	44.19
MapleCB-all-min	236 (141, 95) +0	4683.92	38.05
MapleCB-all-act	240 (141, **99**) +4	4626.99	41.16
MapleCB-all-inclusive	240 (**142**, 98) +4	4602.13	**37.52**

Fig. 5. Benchmark results of 1-UIP, *pure-alluip*. *min-alluip*, *alluip-active* and *alluip-inclusive* on SAT2019 race main track instances.

SAT 2019 Winner *MapleCB-DL*	238 (140, **98**)	4531.24	60.91
MapleCB-DL-all-pure	240 (142, **98**) +2	4519.08	43.32
MapleCB-DL-all-min	244 (**148**, 96) +6	**4419.84**	**36.88**
MapleCB-DL-all-act	243 (146, 97) + 5	4476.73	40.65
MapleCB-DL-all-inclusive	**243** (**148**, 95) +5	4455.76	37.02
SAT 2019 Competitor *expMaple*	237 (137, 100)	4628.96	63.19
expMaple-all-pure	235 (136, 99) −2	4668.96	48.26
expMaple-all-min	241 (143, 98) +4	4524.28	46.29
expMaple-all-act	244 (143, **101**) +7	**4460.92**	47.25
expMaple-all-inclusive	**245** (**146**, 99) +8	4475.76	**45.33**
*CaDiCaL*version 1.2.1 *CaDiCaL-default*	249 (150, 99)	**4311.76**	101.96
CaDiCaL-default-all-pure	248 (151, 97) −1	4373.38	82.44
CaDiCaL-default-all-min	248 (149, 99) −1	4398.31	43.93
CaDiCaL-default-all-act	**252** (152, **100**) +3	4331.56	47.34
CaDiCaL-default-all-inclusive	251 (**153**, 98) +2	4335.88	**42.61**

Fig. 6. Benchmark results of 1-UIP, *pure-alluip*. *min-alluip*, *alluip-active* and *alluip-inclusive* on SAT2019 race main track instances.

alluip-active, *alluip-inclusive*, and *alluip-exclusive*. We chose *CaDiCaL* because its default configuration (*CaDiCaL-default*) solved the most instances in the 2019 SAT Race (244). For this experiment, we used the latest available version of *CaDiCaL-default* instead of the 2019 SAT Race version [3]. We compared these solvers' base 1-UIP learning scheme with *pure-alluip*, *min-alluip* and the top two *stable-alluip* variants, *alluip-active* and *alluip-inclusive*, on the SAT Race 2019 main track benchmarks. We report solved instances, PAR-2 score and the average clause size.

Figures 5 and 6 show the results of the *stable-alluip* configurations in our suite of 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 (+3.9 instances in average) and PAR-2 score (−57.7 in average). The average clause size reduction is consistent across all solvers. Each configuration also exhibits different strengths: *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* solved more instances than 1-UIP learning. *pure-alluip* solved more UNSAT and SAT instances while the other configurations improved on solving SAT instances. The clause size reduction of *stable-alluip* is more significant on this solver than on *MapleCOMSPS-LRB*. The SAT 2018 race winner *MapleCB* uses chronological backtracking (CB); three out of four configurations outperformed 1-UIP. On the SAT 2019 race winner *MapleCB-DL*, all four *stable-alluip* configurations solved more instances than 1-UIP. *MapleCB-DL* prioritizes clauses that are learned multiple times. We observed that *stable-alluip* clauses are less likely to be duplicated. As an example, *min-alluip* on 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 more instances than 1-UIP learning. We noticed that both *alluip-active* and *alluip-inclusive* show 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.

CaDiCaL-default with 1-UIP solved 249 instances. Applying *alluip-active* and *alluip-inclusive* helped the solver solve 3 and 2 more instances, respectively. The 1-UIP clauses in *CaDiCaL-default* were much larger than other solvers on average (101 vs 60) but the *stable-alluip* configurations yielded similar clause sizes.

5 Conclusion

In this paper we introduced a new clause learning scheme, *stable-alluip*, that preserves the strengths 1-UIP learning while learning shorter clauses. We provided empirical evidence that using *stable-alluip* and its variants in modern CDCL solvers achieves significant clause reduction and yields useful performance gains.

Our scheme extends 1-UIP learning by performing further resolution beyond the deepest decision level in an attempt to find the UIP at each level in the learnt clause. Since resolutions may increase the clause's LBD by introducing literals from new decision levels, we presented two methods to block such literals from entering the clause. Although our learning scheme is conceptually simple, and we presented optimizations to reduce and balance the learning cost. We additionally presented variants of our schemes to account for features used in state of the art solvers, e.g., local branching heuristics and chronological backtracking.

Although the field of SAT solving has converged on using the 1-UIP learning scheme, we have shown the possibility of developing an effective alternative through understanding the strengths and weaknesses of 1-UIP and clause learning schemes. Our learning scheme can be generalized and further improved by exploring more fine-grained trade-offs between different clause quality metrics beyond clause size and LBD. We also plan to study the interaction between clause learning and variable branching. Since most of the branching heuristics are tailored for 1-UIP scheme, their interactions with other learning schemes requires further study.

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, 11–17 July 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. Biere, A.: CADICAL at the SAT race 2019. In: Heule, M.J.H., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT Competition 2019 Solver and Benchmark Descriptions. University of Helsinki (2019). http://hdl.handle.net/10138/306988
4. Biere, A.: Cadical SAT solver (2019). https://github.com/arminbiere/cadical
5. Bayardo Jr, R.J., 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, 27–31 July 1997, Providence, Rhode Island, USA, pp. 203–208. AAAI Press/The MIT Press (1997). http://www.aaai.org/Library/AAAI/1997/aaai97-032.php
6. Katebi, H., Sakallah, K.A., Marques-Silva, J.P.: Empirical study of the anatomy of modern SAT solvers. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 343–356. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21581-0_27
7. 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
8. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 123–140. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_9
9. Liang, J.H., Oh, C., Ganesh, V., Czarnecki, K., Poupart, P.: Maple-COMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB. In: Balyo, T., Heule, M.J.H., Järvisalo, M.J. (eds.) Proceedings of SAT Competition 2016 Solver and Benchmark Descriptions. University of Helsinki (2016). http://hdl.handle.net/10138/164630
10. 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, 19–25 August 2017, pp. 703–711 (2017). ijcai.org, https://doi.org/10.24963/ijcai.2017/98

11. Chowdhury, M.S., Müller, M., You, J.H.: Four CDCL SAT solvers based on exploration and glue variable bumping. In: Heule, M.J.H., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT Competition 2019 Solver and Benchmark Descriptions. University of Helsinki (2019). http://hdl.handle.net/10138/306988
12. Möhle, S., Biere, A.: Backing backtracking. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 250–266. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_18
13. 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, 18–22 June 2001, pp. 530–535. ACM (2001). https://doi.org/10.1145/378239.379017
14. Nadel, A., Ryvchin, V.: Chronological backtracking. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 111–121. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_7
15. 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
16. Ryvchin, V., Nadel, A.: Maple_LCM_Dist ChronoBT: featuring chronological backtracking. In: Heule, M.J.H., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT Competition 2018 Solver and Benchmark Descriptions. University of Helsinki (2018). https://hdl.handle.net/10138/237063
17. Sabharwal, A., Samulowitz, H., Sellmann, M.: Learning back-clauses in SAT. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 498–499. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_53
18. 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
19. 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, 10–14 November 1996, pp. 220–227. IEEE Computer Society/ACM (1996). https://doi.org/10.1109/ICCAD.1996.569607
20. Silva, J.P.M., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999). https://doi.org/10.1109/12.769433
21. Sörensson, N., Biere, A.: Minimizing learned clauses. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 237–243. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_23
22. Kochemazov, S., Zaikin, O., Kondratiev, V., Semenov, A.: MapleLCMDistchronoBT-DL, duplicate learnts heuristic-aided solvers at the SAT race 2019. In: Heule, M.J.H., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT Competition 2019 Solver and Benchmark Descriptions. University of Helsinki (2019). http://hdl.handle.net/10138/306988

23. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31
24. Wieringa, S., Heljanko, K.: Concurrent clause strengthening. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 116–132. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_10
25. 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, 4–8 November 2001, pp. 279–285. IEEE Computer Society (2001). https://doi.org/10.1109/ICCAD.2001.968634