DPLL with restarts linearly simulates CDCL

Olivier Bailleux

DPLL with restarts linearly simulates CDCL

Olivier Bailleux
LIB, Université de Bourgogne
olivier.bailleux@u-bourgogne.fr

Research report, January, 2019

Abstract

If we give DPLL the ability to make restarts and to learn clauses representing the explored search space, it can linearly simulate CDCL solvers.

1 Introduction

Second generation SAT solvers, such as DPLL, and third generation SAT solvers, known as CDCL, have several common features. They produce sequences of assumptions (often called decisions) and unitary propagations that lead either to the discovery of a model or to a contradiction, also called conflict. In the following, such a sequence will be called AUP-sequence, for Assumption and Unit Propagation Sequence, and inconsistent AUP-sequence when it leads to a conflict (an example is given in figure 1). But the two types of solvers do not exploit the contradictory AUP-sequences in the same way.

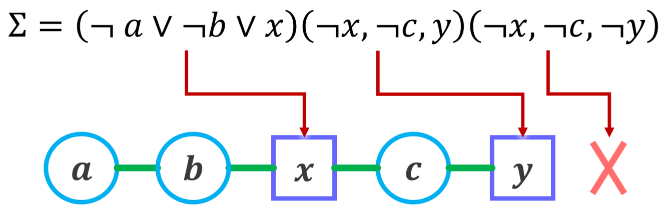
$\Sigma = (\neg a \vee \neg b \vee x)(\neg x, \neg c, y)(\neg x, \neg c, \neg y)$
a
b
x
c
y

Figure 1: An example of a contradictory AUP-sequence. Circles refer to assumptions and squares to propagations.

Third-generation solvers analyze the sequence backwards, from the end, to seek an explanation of the conflict in the form of a clause that will be learned [KSMS11], and then perform a non-chronological backtrack to the stage in the sequence where the learned clause becomes assertive (allows a deduction by unit resolution). Instead, the second generation solvers simply perform a chronological backtrack, which amounts to *implicitly* learn the clause representing the negation of the assumptions. In addition, third generation solvers perform restarts while keeping all or part of the learned clauses. In practice, restarts are crucial for the effectiveness of these solvers. They have also been used with second-generation SAT solvers [GSK98] [SI09], but, until further notice, without preserving any information about the search space explored in previous sessions.

Let us define a *UP-nogood* of a formula Σ as any set E of literals such that the conjunction of Σ and the literals of E allows unit propagation to produce the empty clause, namely $\Sigma \wedge_{w \in E} w \vdash^{UP} \bot$.

The two types of solvers generally do not learn the same clauses. DPLL solvers implicitly use ad absurdum reasoning and implicitly produce the UP-nogood consisting of the assumptions of the current contradictory AUP-sequence, and then implicitly learn the clause corresponding to the negation of this nogood. In addition, the assumptions are stated in an order imposed by chronological backtracking. In the case of CDCL solvers, the UP-nogood is produced from the current contradictory AUP-sequence by starting from the negation of the clause that became empty, and by iteratively replacing a literal w of this nogood by the literals that allowed to deduce w. In most CDCL solvers, this process stops when exactly one of the literals in the current nogood have been fixed at the last assumption level. This learning scheme is known as first-UIP for First Unit Implication Point. For more details, see for example [KSMS11].

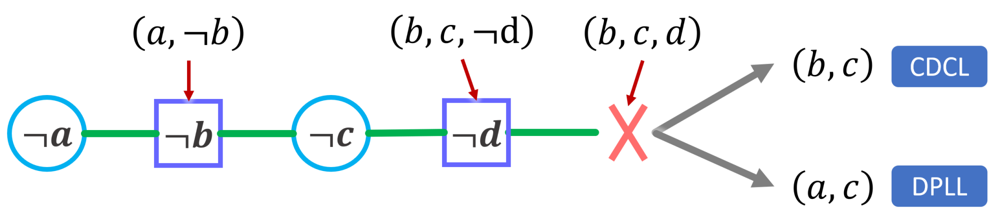

Figure 2: On this example of a contradictory AUP-sequence, DPLL *implicitly* produces the clause $a \vee c$, while CDCL explicitly produces the clause $b \vee c$.

CDCL solvers are known to have the same deductive power as General Resolution in the sense that they can produce refutations whose size is polynomially related to that of the shortest refutations that can be produced by Resolution [PD11]. This is not the case with DPLL solvers, which can refute only in exponential time some formulae that can be refutable in polynomial time and space by Resolution, and therefore by CDCL solvers, subject to an appropriate choice of assumptions [BSIW04]. This applies in particular to Pebbling formulas, which encode the possibility of performing a set of tasks subject to precedence constraints.

The common feature of second and third generation solvers is that when they deal with inconsistent formulae, they implicitly produce refutations of these formulae in a particular form called RUP for "Reverse Unit Propagation" [GN03]. An RUP of a CNF formula Σ is a sequence $q_1, ..., q_m$ of clauses such that for any $i \in 1...m$, $\Sigma \wedge q_1 \wedge ... \wedge q_{i-1} \wedge \neg q_i$ can be refuted by unit propagation (formally, $\Sigma \wedge q_1 \wedge ... \wedge q_{i-1} \wedge \neg q_i \vdash^{UP} \perp$), from which it follows that each clause q_i is a logical consequence of Σ. Each clause of an RUP is therefore a key to produce, thanks to unitary propagation, a step of a proof establishing that the last clause of the sequence is a logical consequence of the formula. See [HB15], for a detailed presentation of the concept of RUP and other variants of clausal proofs.

For example,

$$q_1 = (b \vee c), q_2 = (b), q_3 = \perp$$

is a RUP of the formula

$$\Sigma = (a \vee \neg b)(b \vee c \vee \neg d)(b \vee c \vee d)(b \vee \neg c \vee e)(b \vee \neg c \vee \neg e)(\neg b \vee e)(a \vee \neg b \vee e)(\neg a \vee \neg b \vee \neg e) \tag{1}$$

because

$$\neg q_1 \wedge \Sigma \vdash^{UP} \perp,$$
$$\neg q_2 \wedge q_1 \wedge \Sigma \vdash^{UP} \perp,$$
$$\neg q_3 \wedge q_2 \wedge q_1 \wedge \Sigma \vdash^{UP} \perp$$

The concept of RUP is essentially used for the verification of refutations produced by SAT solvers when they deal with inconsistent formulas, particularly during competitions, where the concerned solvers must produce RUP refutations that are checked by an independent application. Actually, DPLL and CDCL solvers can be considered as machines for producing RUPs. By adopting such a point of view, CDCL's strength is to produce, with certain formulas, RUPs that are much smaller than the smallest RUPs that can be produced by DPLL solvers.

The purpose of this paper is to present a version of the DPLL with restart that learns, before each restart, a set of clauses representing the knowledge accumulated during the previous search session. We will call such a program DPLL+R (for DPLL + Restart) or 2.5th *generation solver*. With a sufficient number of restarts, and by choosing the assumptions appropriately, such a solver can produce exactly the same RUPs as CDCL solvers. In practice, this does not guarantee that such DPLL+R solvers are as effective as CDCL solvers, but it can be argued that the difference in efficiency depends only on the restart policy and the heuristic of assumption selection. This opens some perspectives for exploring the research field between second generation and third generation solvers. In addition, this approach has already been successfully tested in the field of constraint programming [GBL+17].

2 DPLL + Restarts

Applied to a formula Σ, DPLL explores the space of all possible assignments of the variables involved in this formula. To do this, it produces a series of contradictory AUP-sequences under a constrained order of assumptions. More precisely, the state of the solver at the time of a conflict occurring after m assumptions is characterized by a list $(h_1, p_1), ..., (h_m, p_m)$ where h_i denotes an assumption represented by a literal, and $p_i \in \{1, 2\}$ represents the *phase* of this assumption in the current sequence. The successive solver states and associated sequences are subject to the following constraints, which reflects the notion of chronological backtracking:

The conflicting state following any conflicting state $(h_1, p_1), ..., (h_{i-1}, p_{i-1}), (h_i, 1), (h_{i+1}, 2), ..., (h_m, 2)$ must be (h_1, p_1), ..., $(h_{i-1}, p_{i-1}), (\neg h_i, 2), (h'_{i+1}, 1), ..., (h'_{m'}, 1)$, i.e., the last assumption in phase 1 must be complemented, then switched in phase 2 and followed, if applicable, by assumptions in phase 1.

Each conflicting state e of the solver characterize a set C_e of counter-models of the interpretation space of the formula Σ. Clearly, if $e = (h_1, p_1), ..., (h_m, p_m)$, then $C_e = C(e)$ where C is the function defined as follows:

$$\begin{cases} C((h,1)) & = & h & & \\ C((h,2)) & = & \top & & \\ C((h_1,1),(h_2,p_2),...,(h_m,p_m)) & = & h_1 & \wedge & C((h_2,p_2),...,(h_m,p_m)), \quad m > 1 \\ C((h_1,2),(h_2,p_2),...,(h_m,p_m)) & = & \neg h_1 & \vee & C((h_2,p_2),...,(h_m,p_m)), \quad m > 1 \end{cases} \tag{2}$$

For example, with a lexicographic order of assumptions, DPLL refutes the previously introduced formula Σ (1) as shown figure 3.

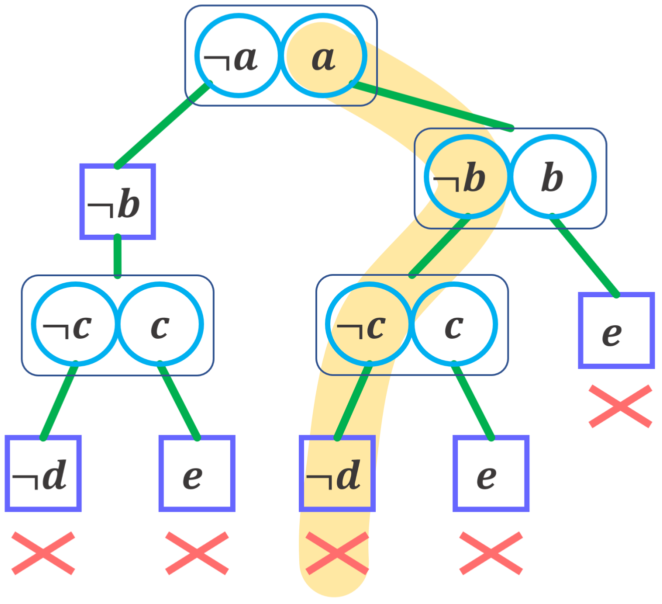

Figure 3: The search tree of DPLL on the formula (1).

At the time of the conflict ending the sequence highlighted in yellow, the solver is in the state $e = (a, 2), (\neg b, 1), (\neg c, 1)$. The interpretations already eliminated at this stage, as defined by the formula (2), are the models of the formula $C_e = \neg a \vee (\neg b \wedge \neg c)$. These interpretations are counter-models of Σ, and represent the knowledge deduced by the solver. This knowledge can be represented by the clauses $(\neg a)$ and $(b \vee c)$.

More generally, the knowledge C_e capitalized by a DPLL session that has reached a conflicting state e can be very easily represented by a set of clauses, i.e., the negation of C_e. Note that the cardinal of this set is linearly related to the number of assumptions in phase 2. The algorithm DPLL+R consists in repeating the following actions until either a model is found or a conflicting state with all the assumptions in phase 2 is reached.

1. Run a DPLL search session that can be stopped (according to a criterion to be defined) at any conflict state.
2. Produce the clauses encoding the knowledge capitalized by the search session using the formula (2).

3 Deduction power of DPLL+R

In this section, we will show that if DPLL+R had a way to choose the assumptions appropriately, it could produce any RUP that could be produced by any CDCL solver, and therefore it could solve any SAT instance as effectively as any CDCL solver. To the best of our knowledge, this result, although quite obvious, has not yet been explicitly mentioned in the literature. It is based on the idea of using an RUP produced by a CDCL solver as an oracle to choose the assumptions of a DPLL solver performing a restart at each conflict (i.e. at the first conflict of each research session).

We will call AUP-nogood of a formula Σ any list $n_1, ..., n_t$ of literals that, when used as assumptions, produce a contradictory AUP-sequence, meaning that all the possible propagations from Σ do not produce any w such that $\neg w \in \{n_1, ..., n_t\}$, all the possible propagations from $\Sigma \wedge n_1 \wedge ... \wedge n_i, i \in 1..t-1$ do not produce any w such that $\neg w \in \{n_{i+1}, ..., n_t\}$, and, the empty clause is propagated from $\Sigma \wedge n_1 \wedge ... \wedge n_t$.

Clearly, the set of the literals of any AUP-nogood of a formula Σ is a UP-nogood of Σ. This is a direct consequence of the confluence theorem of domain consistency in constraint programming [Apt03]. But the reverse is not true. Some UP-nogoods are not AUP-nogoods, first of all, the negation of a clause of Σ. For example, if $(a \vee b \vee c)$ is a clause of Σ, and $\{\neg a, \neg b\}$ is not a UP-nogood of Σ, then $< a, b, c >$ is not a AUP-nogood because $\neg a \wedge \neg b$ will propagate c, and therefore $\neg c$ cannot be used as the next assumption.

In the following, we will show that any UP-nogood produced by a CDCL solver, when sorted appropriately, is an AUP-nogoods. As a result, DPLL+R can simulate CDCL by using as assumptions, in the appropriate order, the literals of this UP-nogood.

For example, the figure 4 shows again the contradictory sequence S_1 from the figure 2 produced from the example formula Σ (1). From this sequence, a CDCL solver produces the AUP-nogood $\neg b, \neg c$ from which the clause $(b \vee c)$ will be learned. This clause will appear in the RUP generated by the CDCL solver. Not all the literals of this AUP-nogood correspond to assumptions in the contradictory sequence from which it was produced. But if we use these literals as assumptions, we produce a contradictory sequence S'_1.

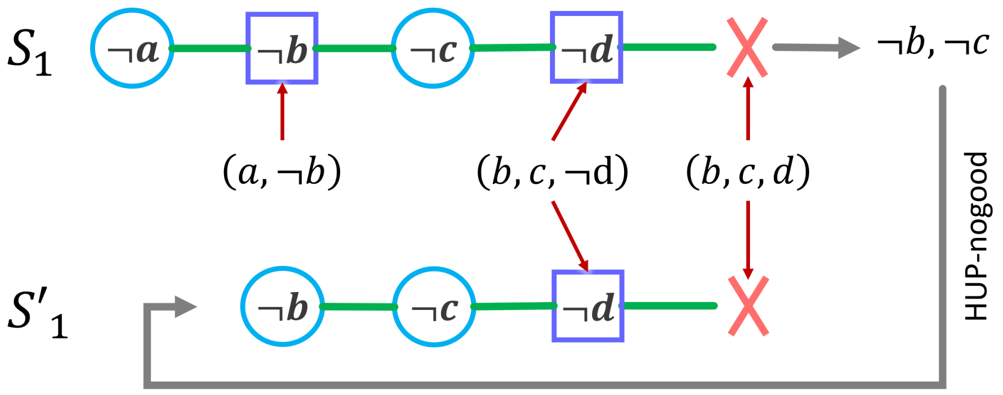

Figure 4: A contradictory AUP-sequence S_1 produced from the formula (1), the AUP-nogood produced by analysis of this sequence by a CDCL solver, and the contradictory sequence S'_1 obtained by using the literals of the nogood as assumptions.

Let us define a *consistent* Σ-AUP-sequence s as a sequence $s_1, ..., s_k$ of *segments*, where each segment s_i consists of a literal h_i called assumption and a sequence L_i of literals verifying the following properties:

1. Neither h_i nor $\neg h_i$ can be deduced by unit propagation from Σ and the assumptions of the previous segments.
2. Each literal of L_i can be deduced by one unit resolution from one or more previous literals of s and a clause of Σ.
3. Any literal that can be deduced by unit propagation from $\Sigma \wedge h_1 \wedge ... \wedge h_i$ is in one of the segments L_1 to L_i.
4. The empty clause cannot be deducted from $\Sigma \wedge h_1 \wedge ... \wedge h_k$ by unit propagation.

Let us define a *contradictory* Σ-AUP-sequence s as a sequence $s_1, ..., s_k, c$ of *segments* such as $s_1, ..., s_k$ is a consistent Σ-AUP-sequence and c is a segment consisting of a literal h_c and a sequence L_c of literals verifying the following properties:

1. Neither h_c nor $\neg h_c$ can be deducted by unit propagation from the assumptions of the previous segments and Σ.
2. Each literal of L_c can be deduced by one unit resolution from one or more previous literals of s and a clause of Σ.
3. The empty clause can be deduced by one unit resolution from the last literal of L_c, one or more previous literals of s and a clause of Σ.

We will only write AUP-sequence when there is no ambiguity about the formula used for deductions. The figure 5 illustrates the composition of a AUP-contradictory sequence.

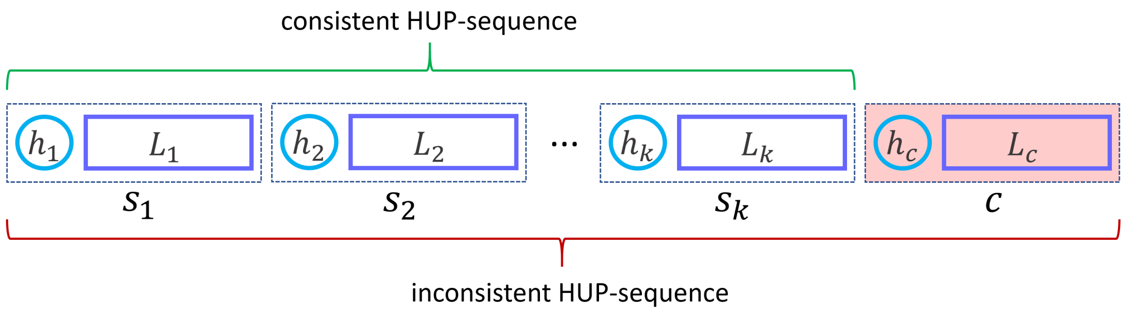

Figure 5: A contradictory AUP-sequence.

Please note that the fact that a Σ-AUP-sequence $s_1, ..., s_k$ is consistent does not tell us that the empty clause is not a logical consequence of $\Sigma \wedge h_1 \wedge ... \wedge h_k$ but tells us that the empty clause cannot be deduced from this formula by unit resolution. On the other hand, if a Σ-AUP-sequence $s_1, ..., s_k, c$ is contradictory, then $\Sigma \wedge s_1 \wedge ... \wedge s_k \wedge c$ is inconsistent, so the clause $\neg h_1 \vee ... \vee \neg h_k \vee \neg h_c$ is a logical consequence of Σ.

We will call the list $h_1, ..., h_k(, h_c)$ of assumptions the *key* of the AUP-sequence $s_1, ..., s_k(, c)$. The sequences that can be generated from the same key can differ in the order of the propagations done in each segment when several unit resolutions are possible. The key of a contradictory AUP-sequence is clearly a AUP-nogood.

Now, we will show that the nogoods produced by CDCL solvers are indeed AUP-nogoods, and therefore can allow DPLL+R to refute a formula as quickly as CDCL if (1) it makes a restart after each AUP-sequence and (2) it uses as assumptions the literals of the nogoods produced by CDCL, in the appropriate order.

Let us recall how CDCL solvers produce their nogoods from a contradictory AUP-sequence $s = s_1, ..., s_k, c$ from a so called *conflict clause* q_c which was become empty at the end of segment c. The production of a nogoods is done in several steps. Each step replaces the current nogood, which is initially the negation of the conflict clause, with a new nogood, until a stopping criterion is verified. Whatever the learning scheme used, at the time of stopping the process there is always exactly one literal of the current nogood in the last segment c of s. In the case of the 1-UIP learning scheme, the most common one, the stop occurs as soon as this condition is fulfilled. Here is the detailed algorithm.

```
Data: a contradictory AUP-sequence $s$
Result: a nogood $N$
$E \leftarrow \{\neg w, w \in q_c\}$, where $q_c$ is the conflict clause of $s$;
while the stopping criterion is false do
    let $r$ be the literal of $E$ that has the highest rank (closest to the end) among the literals of $E$ that are not in
      the key of $s$ (i.e. among those that have been deducted by unit propagation);
    let $q$ be the clause that was used to produce $r$;
    $E \leftarrow E \backslash \{r\} \cup \{\neg w, w \in q \backslash \{r\}\}$ ;
end
$N \leftarrow$ the elements of $E$ sorted according to their ranks in $s$;
```
Algorithm 1: Production of a sorted nogood.

Now, let us introduce the following property.

Proposition 3.1. *Let Σ be a CNF formula, $s = s_1, ..., s_k, c$ be a Σ-AUP-sequence, and $N = n_1, ..., n_t$ be a nogood produced from s using algorithm 1, where $n_1, ..., n_t$ are sorted according to their positions in s. For each $i \in 2..t-1$, $\neg n_i$ cannot be propagated from $n_1, ..., n_{i-1}$, namely, $\Sigma \wedge n_1 \wedge ... \wedge n_{i-1} \nvdash^{UP} \neg n_i$.*

Proof. Let $n_i, i \in 1..t$ be a literal of N. We know that n_i is in s. The tree following cases can occur :

1. n_i is an assumption of s. In this case, if $\neg n_i$ could have been propagated from the previous literals of N, which represent a subset of the literals located before n_i in s, then $\neg n_i$ would have been propagated in s, (because, by construction of s, all possible propagations are achieved before each assumption), and then n_i could not be an assumption of s. The property holds.

2. n_i is propagated in the last segment of s. In this case, $i = t$ and all the literals $n_j, j \in 1..t-1$ of N are in the consistent part of s. If these literals had allowed the propagation of $\neg n_i$, this propagation would have been done in the consistent part of s and n_i would therefore not have been propagated in the last segment. The property holds.

3. n_i is propagated in a segment s_j of the consistent part of s. The literals before n_i in N are located in s either in the segments before s_j, or in s_j. But if these literals had been able to propagate $\neg n_i$, they would have done no latter than in the segment s_j, and the propagation of n_i would have produced the empty clause in the segment s_j. This is in contradiction with the fact that s_j is in the coherent part of s. The property holds.

□

As a corollary, the UP-nogoods produced by CDCL, if sorted in the appropriate order, are AUP-nogoods. It follows that DPLL+R can linearly simulate any refutation of a formula performed by a CDCL solver, at the cost of a restart after each contradictory AUP-sequence, and subject to the choice of the appropriate assumptions (decision variables).

4 Conclusion

Giving a DPLL solver the ability to restart and learn clauses representing the knowledge acquired since the last restart gives it the same deductive power as a CDCL solver. This does not necessarily imply that in practice there are heuristics of assumptions choice that allow DPLL+R to be as effective as CDCL with 1-UIP learning scheme, but it tells us that the difference in efficiency between the two, if applicable, is heuristic in nature. More specifically, the combination of the 1-UIP learning scheme and non-chronological backtracking is not in itself a fundamentally more powerful proof system than the learning of the negation of decision literals (which we have preferred to call assumptions in this article). Rather, it is an effective heuristic for producing refutations by inverse unit propagation (RUP). The question that remains open is the existence of assumption choice heuristics that would allow DPLL+R solvers to produce RUP refutations as short as those produced by the best CDCL solvers.

Although we have exploited, for the purpose of proving the deductive power of DPLL+R, the properties of the borderline case where the solver restarts after each conflict, we think it would be very interesting to experiment with this type of solver with a less drastic restart regime, seeking a good compromise between the theoretical deductive power and the growth of the number of clauses learned, so as to exploit the ability of chronological backtracking to represent in a very concise way the explored part of the interpretation space. The potential of this approach is based on the existence of heuristics of assumption choice that can exploit the theoretical potential of the underlying proof system.

References

[Apt03] Krzysztof R. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

[BSIW04] Eli Ben-Sasson, Russell Impagliazzo†, and Avi Wigderson‡. Near optimal separation of tree-like and general resolution. *Combinatorica*, 24(4):585–603, Sep 2004.

[GBL$^+$17] Gael Glorian, Frederic Boussemart, Jean-Marie Lagniez, Christophe Lecoutre, and Bertrand Mazure. Combining nogoods in restart-based search. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming*, pages 129–138, Cham, 2017. Springer International Publishing.

[GN03] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10886–, Washington, DC, USA, 2003. IEEE Computer Society.

[GSK98] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[HB15] Marijn Heule and Armin Biere. Proofs for satisfiability problems. *All about Proofs, Proofs for All (APPA), Mathematical Logic and Foundations*, 55:1 – 22, 2015.

[KSMS11] Hadi Katebi, Karem A. Sakallah, and Joao Marques-Silva. Empirical study of the anatomy of modern sat solvers. In *SAT*, 2011.

[PD11] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers as resolution engines. *Artificial Intelligence*, 175(2):512 – 525, 2011.

[SI09] Carsten Sinz and Markus Iser. Problem-sensitive restart heuristics for the dpll procedure. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 356–362, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.