Lecture 7— SAT solvers in theory and practice

Massimo Lauria — lauria.massimo@gmail.com
Office 1107, Ookayama West 8th Building

Tuesday — November 17th, 2015 (This document was updated on June 21, 2017)

We discuss concrete SAT solver software. We will comment about their applications and about their performance in practice. We will discuss some formal connection between theoretical models of CDCL solvers and resolution complexity. I will briefly mention some facts about non-resolution based SAT solvers.

Pictures in this lecture come from Marijn Heule slides.

For a long time most SAT solving was based on decision tree. A SAT solver would pick an unassigned variable, set it to some value, and continue on the restricted formula.[1] The algorithm would use clever heuristics in order to reduce as much as possible the final size of the tree. One obvious heuristic that is used massively in all solvers up to date, the *unit propagation*:

> If the CNF F has unit clauses (i.e. clauses that contains only literal each) the next decision is to set one of those literals to true.

[1] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962

This has been the state of the art for SAT solving up to the end of the nineties. A major breakthrough was obtained when solvers realized how to learn new clauses during the exploration of the search space. Now SAT solver can solve formulas with millions of variables and clauses, mostly coming from industrial applications as

- artificial intelligence;
- software static analysis;
- cryptography;
- hardware verification;
- and many more...

A good source of information about CDCL solvers in Chapter 4 of the Handbook of Satisfiability.[2]

[2] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009

Conflict Driven Clause Learning (CDCL) solvers

The solver uses data structure for (1) a stack (2) clause database. At the beginning the database contains the clauses of the formula, the stack is empty.

A CDCL solver decides variable assignments and unit propagates them until a clause is falsified, at which point a clause is learned from the *conflict* and is added to the clause database and the search backtracks.

The assignments are saved on the stack. Each assignment has a decision level. The first unit propagations are at decision *decision level* 0, then the

first variable decided by the solver initiate decision level 1, and all following unit propagations are at level 1 themselves. The next decision opens decision level 2, and so on.

Remark 1. *It is very important to stress that practically all CDCL solvers are greedy, in the sense that if there is a unit propagation available, that unit is propagated, and that if there is some conflict (i.e. the assignment on the stack falsifies a clause in the database) then conflict analysis starts immediately.*

The state of the solver is identified by the state of the stack and of the clause database, and by the mode of operation. It is convenient to describe the solver as being in one of the four modes **Default** (where it starts), **Unit**, **Conflict**, or **Decision**, where transitions are performed as described below.

Default If all variables in the formula are assigned, the solver halts and outputs `SAT`. If the stack falsifies a clause in the database, the solver moves to **Conflict**. Otherwise if the assignment on the stack produces a unit propagation, the solver transits to **Unit** mode. If none of the above cases apply, the solver uses its *restart policy* to decide whether to empty the stack, and its *clause database reduction policy* to decide whether to shrink remove clauses from the database. After this, it then moves to **Decision**.

Conflict If stack has only decision level 0, the solver outputs `UNSAT`. Otherwise the solver applies the *conflict analysis* to derive learn a clause and then *backjumps* (i.e. unrolls some decision levels from the stack) and goes to **Default**.[3]

Unit The solver picks a clause that has all but one literals falsified on the stack and add to the stack that the remaining literal is set to true. Then moves to **Default** mode.

Decision The solver uses the *decision scheme* to determine an assignment with which to extend the trail and moves to **Default** mode.

The state of the solver is *stable* if it makes the solver move from **Default** mode to **Decision** mode and a is a *conflict* if it causes a move from **Default** to **Conflict**. The solver implementation need to specify few components.

- Decision scheme: how to pick the next variable and the next value? We will discuss this later.
- The order of the unit propagations in case there are multiple ones. Usually implementation dependent.
- Restart policy: when to force a flush of the stack. We won't discuss this.
- Clause database reduction policy: when and how remove clauses from the database.[4] We won't discuss this.
- the conflict analysis. We will discuss it next.

[3] In essentially all conflict analysis techniques used in practice, at this point the learned clause immediately causes a unit propagation. We will see later techniques for conflict analysis like that.

[4] A SAT solver would fill the memory of even a powerful machine in seconds, so it needs to clean up the database very aggressively.

Conflict analysis and clause learning

How do we learn clauses? While the solver decide variables and propagates unit, it also produced a *conflict graph*, a directed acyclic graph where nodes correspond to assignments, where each node is labeled by its decision level. Decision have indegree zero and each unit propagation has an incoming edges from all the assignments that triggered that unit propagation. **Conflict:** at some point a clause would unit propagate an assignment $x = b$ when instead $x = \neg b$ is on the stack. This is a conflict. The solver stops and analyze the implication graph to learn a clause (see Figure 1).

1. We find a cut of the graph so that the conflict is on the right side[5], and all decisions are on the left side;
2. we take the partial assignment ρ made by all the variables assignments that have an outgoing edge crossing the cut;
3. we add to the clause obtained by negating ρ to the clause database;
4. we remove one or more decision levels from from the stack.

[5] We assume the graph is oriented from left to right.

Point (1) and (4) are the thing that depend on the *clause learning scheme*. Most solver use assertive schemes (explained later) and in particular the 1UIP scheme.

Exercise 2. Prove that the learned clause can be derived, in resolution, from all clauses in the database. What is the shape of this resolution derivation?

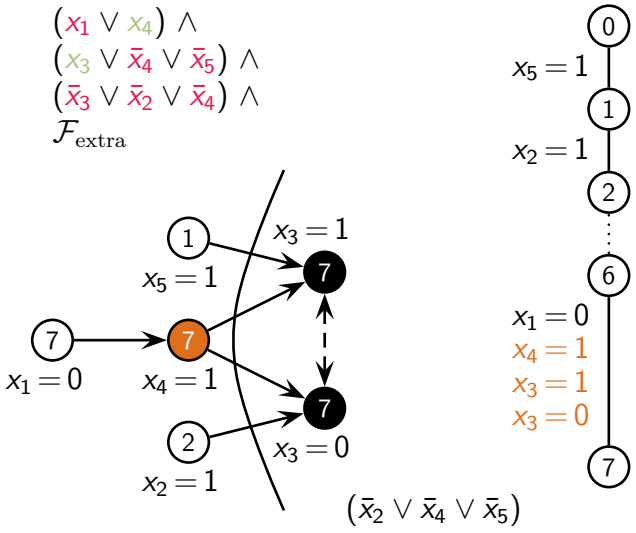

Figure 1: The conflict analysis (From the slides of Marijn Heule)

Unique implication points and First UIP

Once we reach a contradiction in the implication graph we need to choose how to analyze the conflict and which clause to learn. We need to decide

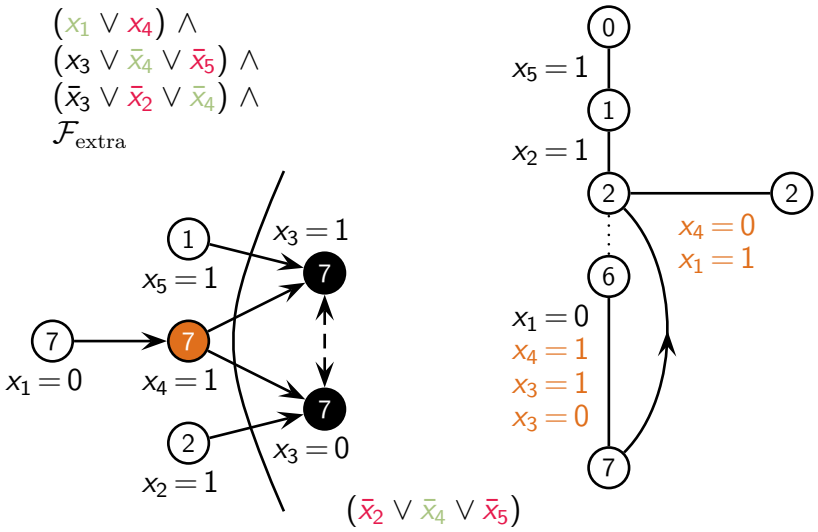

Figure 2: Backjump (From the slides of Marijn Heule)

which clause to learn and to add to the database. In the example in Figure 2 we see that immediately after the backjump there is a unit propagation triggered by the clause just leaned. This is not an accident. Most of the time the learning scheme is tuned to obtain precisely this phenomenon. In Figure 3, 4 and 5 we see an example from Marques-Silva and Sakallah (1999)[6] we the same implication graph provides different clauses, depending on which cut is chosen. In Figure 3 we use the latest possible cut, which gives a clause that contains three literals from the latest decision level. It is more practical instead to use an *asserting clause*.

[6] João P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999

Definition 3. *An asserting clause in a conflict graph is a clause that contains only one literal from the latest decision level. Such literal is called* asserting literal. *The latest decision level among the literals of a clause is called the* asserting level. *An* asserting learning scheme *learns an assertive clause and then backjumps to the asserting decision level.*

This naming is due to the following fact.

Proposition 4. *After the backjump, the asserting literal is immediately unit propagated.*

One particular way to implement an assertive learning scheme looks a cut where exactly one node from the last decision level has an edge that crosses the cut. Each such node is called *unique implication point (UIP)*, and there may be several of them. The implication graph in Figures 3, 4 and 5 has three unique implication points. Most CDCL solvers pick the first one (1UIP)[7] which is one closest to the conflict (see Figure 4), but there are other options as well (see Figure 5). It seems that the first implication point works well in practice, but this is just hands-on experience.

[7] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001

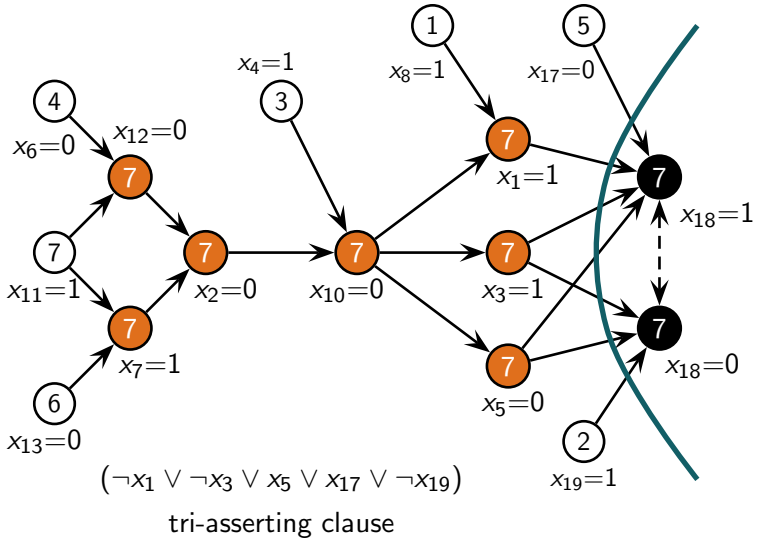

Figure 3: Which clause to learn? (From the slides of Marijn Heule)

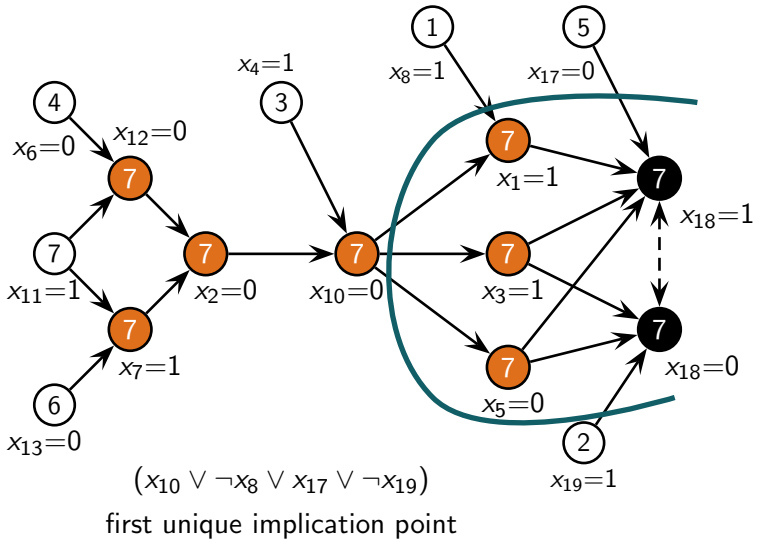

Figure 4: First UIP (From the slides of Marijn Heule)

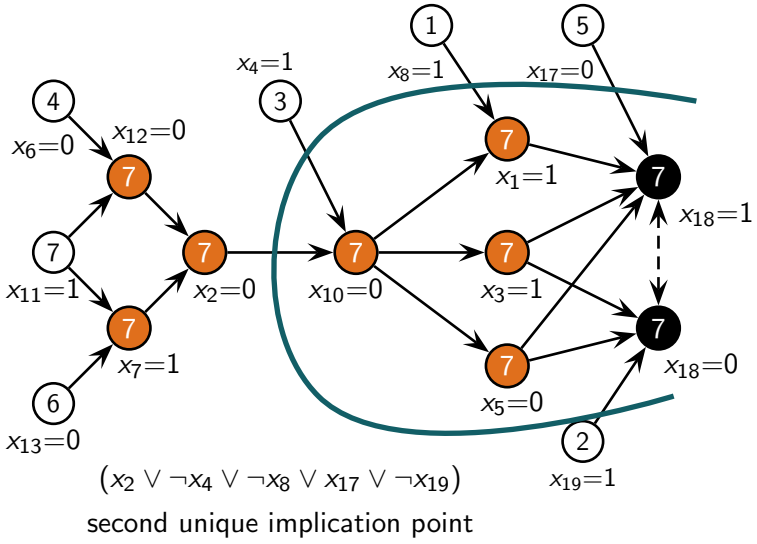

Figure 5: Second UIP (From the slides of Marijn Heule)

Watched literals

A CDCL solver spend almost 100% of its running time performing unit propagations. Therefore it is essential to make them fast. Once a literal is falsified, in theory all clauses containing it would need to be updated. It is expensive to do that. The idea is to keep only two *watched literal* per clause.[8] A clause C that contains two or more literals has two of them highlighted. When a literal is falsified by the decision procedure then instead of updating all clauses that contain that literal, we just update the ones for which the literal is "watched".

[8] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001

- First we look for another unassigned literal to watch;
- if the clause is actually satisfied, we skip it;
- if there is no such literal then it means that there only one unsassigned literal and that it is time to do unit propagation.

Decision scheme: VSIDS

The decision scheme is used to choose the next variable to assign when there are no unit propagations. The dominant decision heuristic in practice is VSIDS, which stands for Variable State Independent Decaying Sum. [9]

We won't discuss so much the details for this heuristic, but just how it works.

[9] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001

1. Each literal has a counter, initialized to 0;
2. when a clause is added to the database, the counter of all involved variables (both polarities) increases;

3. if the solver needs to make a decision, picks the literal with highest score;[10]
4. periodically all counter are scaled by a constant < 1, e.g. 0.9.

[10] Ties broken according to implementation.

The idea is that the solver should focus on variables involved in the last conflicts, and older operation should weight less.

Other improvements

While moving to CDCL solver produced a huge qualitatively jump in performances, there have been other improvements. While these improvements are more specialized and sometimes are even detrimental to the runtime, are very useful in practice.

- Preprocessing: the formula is analyzed and transformed before the CDCL solver starts operating;
- In-processing: a fast and quick form of processing can be done even to the clause database, while the CDCL algorithm run;
- detection and special management of parity and cardinality constraints;
- many others...

Unfortunately many of these improvements cannot be modeled in resolution, and indeed in some cases they were introduces to manage formulas that were theoretically difficult even for general resolution, and that then become easy with this improvements.

Do SAT solver find good resolution proof?

We already discussed how the clause learned by the CDCL solver are deducible in resolution. The connection holds in the other direction, but we need to make strong assumption since resolution proofs are non-deterministic, while an arbitrary CDCL solver could use arbitrarly bad heuristic. The greedy approach of the solver also gets in the way.

In (Pipatsrisawat and Darwiche, 2011)[11] they show that if we grant non-deterministic decision heuristic then the solver finds a refutation of size comparable with the shortest one.

In (Atserias, Fichte, Thurley, 2011)[12] they assume that the learning scheme is computed by the solver itself, therefore it is impossible to find the shortest proof. But if the width of the proof is w and the decision heuristic is sufficiently random, then the solver finds a refutation in about n^k steps.

The two results use very similar proof strategies, and indeed they essentially prove the same result under different perspectives.

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

[12] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artif. Intell. Res. (JAIR)*, 40:353–373, 2011

Theorem 5 (Atserias, Fichte, Thurley, 2011)**.** *A CDCL sat solver with "sufficiently" random decision heuristic refutes, with probability at least* $1/2$ *any CNF on* n *variables having a resolution refutation of width* k *and length* m *in time* $m^{O(1)} \cdot n^{k+1}$*, under any asserting learning scheme.*

Theorem 6 (Pipatsrisawat and Darwiche, 2011)**.** *A CDCL sat solver with non-deterministic decision heuristic refutes any CNF having a resolution refutation of length m in time $m^{O(1)}$, under any asserting learning scheme.*

These theorems have a problem, when used to analyze CDCL solvers. One small issue is that they rely on very frequent restarts, which is not exaclty realistic for many solvers. But the worst issue is that they hold only if the clause database is never cleaned up and if no clause is forgotten. This is definitely unrealistic. Memory is one of the major concern in CDCL SAT solving.

References

[AFT11] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artif. Intell. Res. (JAIR)*, 40:353–373, 2011.

[BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.

[MMZ+01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[MSS99] João P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

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

[ZMMM01] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.