

# Automated Theorem Proving – *Foundations of SAT-Solving* –

July 2019

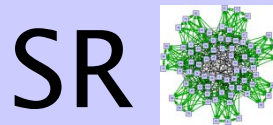
Wolfgang Kuechlin

Symbolic Computation Group  
Wilhelm-Schickard-Institute of Informatics  
Faculty of Mathematics and Sciences

Universität Tübingen

Steinbeis Transferzentrum  
Objekt- und Internet-Technologien (OIT)

[Wolfgang.Kuechlin@uni-tuebingen.de](mailto:Wolfgang.Kuechlin@uni-tuebingen.de)  
<http://www-sr.informatik.uni-tuebingen.de>



# Contents

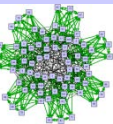
---

## ➤ Propositional Resolution

- Theorem Proving by Deduction, based on *clauses*
- Clause: disjunction of literals, e.g.  $(x \vee \neg y \vee z) \equiv \{x, \neg y, z\}$
- From 2 clauses  $C_1$  and  $C_2$ , deduce new resolvent clause  $R$  where  $C_1 \wedge C_2 \models R$ , hence  $\{C_1, C_2\} \equiv \{C_1, C_2, R\}$
- Prove  $\text{SAT}(\mathcal{F}) = \text{false}$  by deducing  $\perp$  (represented by unsatisfiable *empty clause*  $\{\} =: \square$ ) since  $\mathcal{F} \equiv \mathcal{F} \cup \{\} \equiv \mathcal{F} \wedge \perp \equiv \perp$

## ➤ Elementary DPLL SAT-Solving

- Historical Davis-Putnam-Logemann-Loveland Method (1962)
- Compute  $\text{SAT}(\mathcal{F})$  *in place* as follows:
  - Try some variable assignment  $x = \text{true}$
  - Compute other variable assignments as immediate consequences
  - Carry on, until success, or backtrack to try  $x = \text{false}$ .



# The Resolution Rule

- (sole) inference rule:

$$\frac{C \cup \{\ell\} \quad D \cup \{\neg\ell\}}{C \cup D} \text{Res}$$

where  $C' = C \cup \{\ell\}$  and  $D' = D \cup \{\neg\ell\}$  are clauses.

The *resolvent*  $R = C \cup D$  is deduced by *resolving parent clauses*  $C'$  and  $D'$  *on the literal  $\ell$* . We write  $\{C', D'\} \vdash R$ .

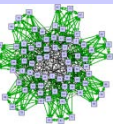
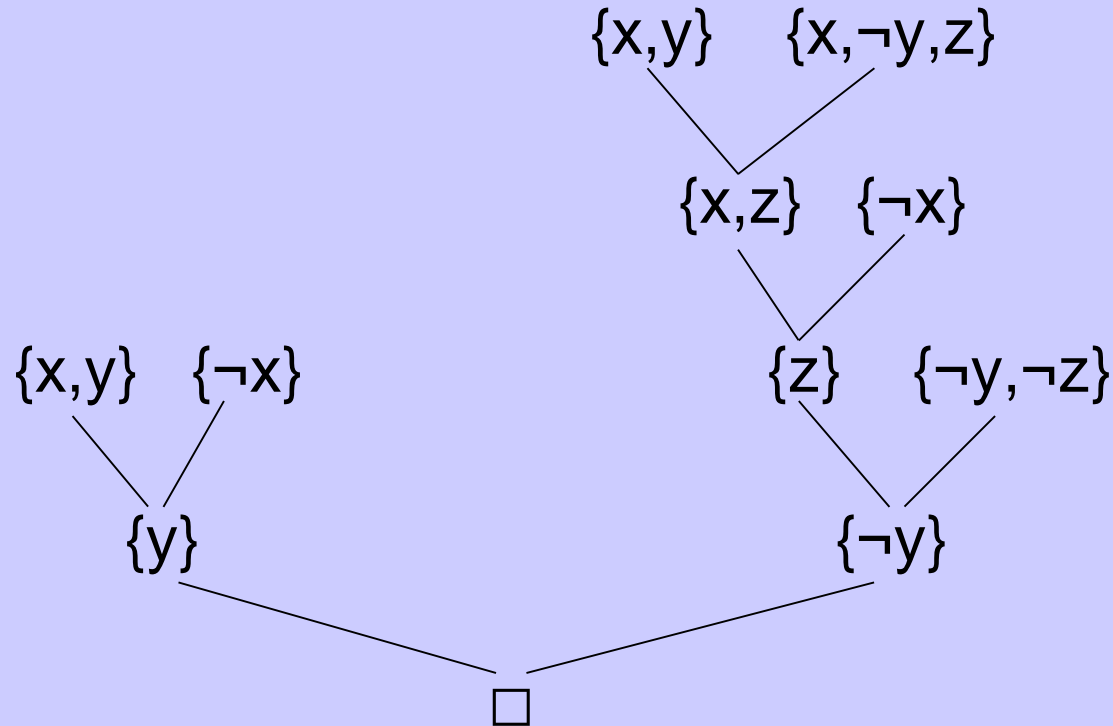
- Examples:

$$\frac{\{x, y, \neg z\} \quad \{u, \neg v, z\}}{\{x, y, u, \neg v\}} \quad \frac{\{x, u, \neg v\} \quad \{\neg u\}}{\{x, \neg v\}}$$



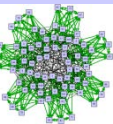
# Example: Resolution proof-tree

$F3 = \{\{x, y\}, \{x, \neg y, z\}, \{\neg x\}, \{\neg y, \neg z\}\}$



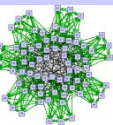
# Clauses and Implications, Soundness, Completeness

- A clause embeds a variety of implications:
  - Example:  $(x \vee y \vee z) \equiv \neg(\neg x \wedge \neg y) \vee z \equiv (\neg x \wedge \neg y \rightarrow z)$
  - But also:  $(x \vee y \vee z) \equiv \neg(\neg x) \vee (y \vee z) \equiv (\neg x \rightarrow y \vee z)$
- Resolution represents deduction of implications
  - Parent clauses  $(A \vee x)$  and  $(B \vee \neg x)$ , resolvent  $(A \vee B)$
  - $(A \vee x) \equiv (\neg(\neg A) \vee x) \equiv (\neg A \rightarrow x)$
  - $(B \vee \neg x) \equiv (x \rightarrow B)$
  - From  $(\neg A \rightarrow x)$  and  $(x \rightarrow B)$  follows  $(\neg A \rightarrow B) \equiv (\neg(\neg A) \vee B) \equiv (A \vee B)$
- The resolution calculus is *sound*. For all sets  $\mathcal{F}$  of clauses and all clauses  $D$ , we have:  $\mathcal{F} \vdash_{\text{Res}}^* D$  implies  $\mathcal{F} \models D$
- Hence:  $\mathcal{F} \vdash_{\text{Res}}^* D$  implies  $\mathcal{F} \equiv \mathcal{F} \cup D$
- Theorem: the resolution calculus is *refutation complete*:  
 $\mathcal{F} \models \perp$  implies  $\mathcal{F} \vdash_{\text{Res}}^* \square$



# Subsumption and Unit Resolution

- Clause  $C$  *subsumes* clause  $D$  iff  $C \subseteq D$ .
  - Constraint  $D$  is subsumed by  $C$ , because  $C$  is stronger.
- If  $C \subseteq D$  then  $C \models D$ .
  - Because  $D \equiv C \vee R$  for some  $R$
- Lemma: Subsumed clauses are redundant and can be cancelled
  - Let  $C \subseteq D$ . Then  $\mathcal{F} \cup C \equiv \mathcal{F} \cup C \cup D$ .
  - Absorption Law:  $F \wedge (F \vee G) \equiv F$ , hence  $C \equiv C \wedge (C \vee R) \equiv C \wedge D$
- In *Unit Resolution* one of the parent clauses is a *Unit* (singleton)
  - For parent clauses  $(A \vee x)$  and  $(\neg x)$ , the resolvent is  $(A)$
  - Unit resolution is valuable, because  $A$  is shorter than  $(A \vee x)$ .
  - Moreover, the (shorter) resolvent subsumes the longer parent:  $A \wedge (A \vee x) \equiv A$
  - Hence the parent can be cancelled



# Resolution proof procedure

➤ Let  $C$  be a set of axioms (constraints). In order to prove a theorem  $D$ , i.e.  $C \models D$ , proceed as follows:

1. Negate  $D$ . Convert  $C \cup \neg D$  into CNF. Call the result  $F$ .

2. Repeat

i. Compute all non-tautological resolvents  $R$  from  $F$

ii. If  $\square \in R$ , return „proof“.

iii. Delete from  $R$  all subsumed clauses. // forward subsumption

iv. If  $R = \{ \}$ , return „disproof“. //  $\square$  not found, no more deductions possible

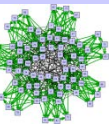
v. Delete from  $F$  all clauses subsumed by  $R$ . // backward subsumption

vi.  $F := F \cup R$

➤ Theorem [refutation completeness]: If  $C \models D$ , the proc. stops in (ii)

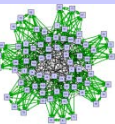
➤ Otherwise the procedure stops in (iv), because there are at most  $3^n$  subsumption-free clauses in  $n$  variables ( $x, \neg x$ , don't care  $x$ ).

➤ Bad news: far too many useless deductions, memory explodes.



# ***SAT-Solving with the DPLL Algorithm***

- DPLL: Davis-Putnam-Logemann-Loveland
  - DPLL dates from 1960/62, significantly improved to CDCL
  - CDCL (conflict driven clause learning) ~1996
- Decision procedure for *SAT-Solving* problem **SAT(F)**
  - Compute SAT(F)=? Is there a satisfying assignment for  $F$ ?
  - Alternative formulation: eliminate quantifiers in  $\exists x_1, \dots, x_n: F$
  - Idea: Iteratively try to construct an assignment, propagating the consequences at each step, backtrack exhaustively if necessary
- CDCL is method of choice for SAT (no contention)
  - Solves „well-behaved“ problems with millions of variables
  - Many „real“ problems are „well-behaved“: circuit and protocol verification, configuration problems, software verification.

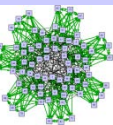




# History: the Davis-Putnam Algorithm (DP-1960)

---

- Martin Davis, Hilary Putnam: A Computing Procedure for Quantification Theory. *Journal of the ACM* 7:201--215 (1960)
  - Context: proving theorems of predicate logic
    - Formula  $P$  is a contradiction iff there is a finite contradiction
    - Iteratively generate the Herbrand universe  $H_i$  and substitute into  $P$
    - Each instance  $P(H_i)$  is a propositional formula, then solve  $\text{SAT}(P(H_i))$
- The DP-1960 algorithm consists of 3 rules
  1. One-Literal Rule (*Unit Propagation – UP*)
  2. Affirmative-Negative Rule (*Pure Literal – PL*)
  3. Elimination of conflicts (*Resolution*)
- Later, rule 3 was replaced by 3\* (DPLL, 1962)
  - 3\*. Splitting Rule (*Case distinction → SAT-Solving*)



# History: the Davis-Putnam Algorithm (DP-1960)

## ➤ Rule 1 (One-Literal Clauses)

- a)  $F$  is inconsistent, if it contains two unit clauses  $\{p\}$  and  $\{\neg p\}$
- b) Else, if  $F$  contains a unit clause  $\{p\}$ , then delete all clauses containing  $p$ , and delete  $\neg p$  from all clauses.

The result  $F'$  is inconsistent iff  $F$  is inconsistent.

- a) The case of a unit clause  $\{\neg p\}$  is analogous to (b).

If  $F'$  is empty, then  $F$  is consistent.

- All clauses were deleted, hence all are satisfied.

## ➤ Rule 2 (Affirmative-Negative Rule)

- If an atom  $p$  appears only positively (affirmative) or only negatively, then delete all clauses containing  $p$ .
- The result  $F'$  is inconsistent iff  $F$  is inconsistent.
- If  $F'$  is empty, then  $F$  is consistent.



# History: the Davis-Putnam Algorithm (DP-1960)

## ➤ Rule 3 (Elimination of Atomic Formulas)

- If an atom  $p$  appears **both** positively (in clause set  $A$ ) **and** negatively (in clause set  $B$ ), then

form clause-sets  $A'$  and  $B'$  with  $A = A' \vee p$  and  $B = B' \vee \neg p$  and rearrange  $F$  into  $F = (A' \vee p) \wedge (B' \vee \neg p) \wedge R$ , where  $p$  does not occur in  $A'$ ,  $B'$  and  $R$ .

Now  $F$  is inconsistent iff  $F' = (A' \vee B') \wedge R$  is inconsistent

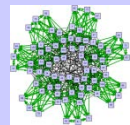
- Proof:

$F$  is inconsistent *iff* it is inconsistent for *both*  $p=0$  and  $p=1$ .

$F = A' \wedge R$  for  $p=0$ , and  $F = B' \wedge R$  for  $p=1$ ,

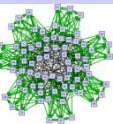
hence  $F$  is inconsistent *iff*

$F' = (A' \wedge R) \vee (B' \wedge R) = (A' \vee B') \wedge R$  is inconsistent.



# History: the Davis-Putnam Algorithm (DP-1960)

- Implementation of Rule 3 (Eliminating Atomic Formulas)
  - Rearrange  $F$  into  $F = (A' \vee p) \wedge (B' \vee \neg p) \wedge R$
  - $F$  is inconsistent iff  $F' = (A' \vee B') \wedge R$  is inconsistent
  - In short: factor out  $p$  and resolve on  $p$ .  $(A' \vee B')$  consists of all resolvents between clauses in  $A$  and clauses in  $B$ .
    - Ex.:  $(a \vee p) \wedge (b \vee p) \wedge (c \vee \neg p) \wedge (d \vee \neg p) = [(a \wedge b) \vee p] \wedge [(c \wedge d) \vee \neg p]$   
Form  $(a \wedge b) \vee (c \wedge d) = (a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$ . These are exactly the resolvents. The parent clauses can be deleted: if  $(a \wedge b)$  is satisfied in  $F'$ , then  $F$  is satisfied by additionally setting  $p=0$ , and analogously, if  $(c \wedge d)$  is satisfied in  $F'$ , then we may set  $p=1$  to satisfy  $F$ .
- Bad news: clauses get longer, and clause set explodes:  $F' = (A' \vee B') \wedge R$  is no longer in CNF, needs fresh CNF conversion



# Example: the Davis-Putnam Algorithm (DP-1960)

## ➤ The DP-Algorithm consists of 3 rules

1. One-Literal (Unit Propagation – UP)
2. Affirmative-Negative (Pure Literal – PL)
3. Elimination of conflicts (Resolution)

## ➤ Example

- $S_0 = \{\{x, y, z\}, \{\neg x, y, z\}, \{\neg x\}, \{z, \neg y\}\}$
- Rule 1c (UP of  $\neg x$ ):  $S_1 = \{\{y, z\}, \{z, \neg y\}\}$
- Rule 3 (resolution on  $y$ ):  $S_2 = \{\{z\}, \{z\}\} = \{\{z\}\}$
- Rule 2 (PL of  $z$ ):  $S_3 = \{ \}$ , hence consistent.

## ➤ Rule 3 renders DP-1960 inefficient

- In order to eliminate  $p$  from  $F$ , *ALL* resolvents over  $p$  have to be computed. This leads to an explosion of new clauses.



# Existential Quantifier Elimination (EQE) with DP-1960

- The DP-Algorithm consists of 3 rules
  1. One-Literal (Unit Propagation – UP)
  2. Affirmative-Negative (Pure Literal – PL)
  3. Elimination of conflicts (Resolution)
- Rules 2 and 3 yield an EQE procedure
  - SAT as an EQE problem:  $\text{SAT}(F) \text{ iff } \exists x_1, \dots, x_n: F \equiv \top$  and  $\text{UNSAT}(F) \text{ iff } \exists x_1, \dots, x_n: F \equiv \perp$
  - Definition:  $\exists p. F \equiv F_{[p/\top]} \vee F_{[p/\perp]}$
  - If  $p$  is pure then  $F = (A' \vee p) \wedge R$ , hence  $\exists p. F \equiv R \vee (A' \wedge R) \equiv R$
  - ... else  $F = (A' \vee p) \wedge (B' \vee \neg p) \wedge R$ , hence  $\exists p. F \equiv (A' \vee B') \wedge R$ .
  - One of the rules is applicable as long as there are literals in  $F$
  - In the end, either  $F = \{ \}$  ( $F$  is SAT) or  $F = \{ \{ \} \}$  ( $F$  is UNSAT)



## Example: EQE with DP-1960

➤ First approach: Use rule 3 until all literals are pure

- Rule 1 is special case of rule 3

➤ Example

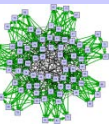
- $S_0 = \{\{x, y, z\}, \{\neg x, y, z\}, \{\neg x\}, \{z, \neg y\}\}$
- Rule 3 (resolution on  $y$ ):  $S_1 = \{\{x, z\}, \{\neg x, z\}, \{\neg x\}\}$
- Rule 3 (resolution of  $x$ ):  $S_2 = \{\{z\}, \{\neg x, z\}, \{\neg x\}\}$
- Rule 2 (PL of  $z$ ):  $S_3 = \{\{\neg x\}\}$
- Rule 2 (PL of  $\neg x$ ):  $S_4 = \{\}$ , hence consistent.

➤ Second approach: Prefer rule 2 over rule 3

- Rule 2 (PL of  $z$ ):  $S_1 = \{\{\neg x\}\}$ ; Now  $\neg x$  is pure and  $S_2 = \{\}$ .

➤ But EQE is *not really* SAT-Solving

- We may not get a satisfying assignment on impure literals



# The Davis-Logemann-Loveland Algorithm (DPLL 1962)

- Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem Proving. *Communications of the ACM* 5:394—397 (1962).
- Rule 3\* (*Splitting Rule*, replaces Rule 3)
  - If  $p$  occurs both positively and negatively in  $F$ , rearrange  $F$  into  $F = (A \vee p) \wedge (B \vee \neg p) \wedge R$ , where  $p$  does not occur in  $R$ .
  - $F$  is inconsistent iff *both*  $(A \wedge R)$  *and*  $(B \wedge R)$  are inconsistent
  - *Proof*:  $F$  is inconsistent iff it is inconsistent for *both*  $p=0$  *and* for  $p=1$ . Now  $F = A \wedge R$  for  $p=0$ , and  $F = B \wedge R$  for  $p=1$ .
- Implementation of Rule 3\*
  - Set  $p=1$  and  $p=0$  *one after another* in  $F$  (e.g. as unit clauses)
  - Do not form new clauses, instead perform unit propagation.
  - Clauses are shortened, sometimes eliminated, the problem is simplified, especially through unit propagation.





# History: The Davis-Logemann-Loveland Algorithm

The forms of Rule III are interchangeable; although theoretically they are equivalent, in actual applications each has certain desirable features. We used Rule III\* because of the fact that Rule III can easily increase the number and the lengths of the clauses in the expression rather quickly after several applications. This is prohibitive in a computer if one's fast access storage is limited. Also, it was observed that after performing Rule III, many duplicated and thus redundant clauses were present. Some success was obtained by causing the machine to systematically eliminate the redundancy; but the problem of total length increasing rapidly still remained when more complicated problems were attempted. Also use of Rule III can seldom yield new one-literal clauses, whereas use of Rule III\* often will.

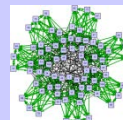
In programming Rule III\*, we used auxiliary tape storage. The rest of the testing for consistency is carried out using only fast access storage. When the "Splitting Rule" is used one of the two formulas resulting is placed on tape. Tape memory records are organized in the cafeteria stack-of-plates scheme: the last record written is the first to be read.



# The DPLL Algorithm (of 1962)

---

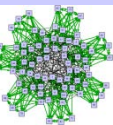
```
boolean DPLL(ClauseSet S){  
  
    //1. Simplify S (unit constraint propagation)  
    while (S contains a unit clause { $\ell$ }) {  
        delete from S clauses containing  $\ell$ ;           // unit-subsumption  
        delete  $\neg\ell$  from all clauses in S           // unit-resolution mit subsumption  
    }  
  
    //2. Trivial case?  
    if ( $\square \in S$ ) return false;                       // constraint unsatisfiable  
    if ( $S == \{\}$ ) return true;                          // nothing left to satisfy  
  
    //3. Case split and recursion  
    choose a literal  $\ell$  occurring in S;                // Heuristic (intelligence) needed!  
    if( DPLL( $S \cup \{\ell\}$ ) ) return true;              // first recursive branch: try  $\ell := \text{true}$   
    else if ( DPLL( $S \cup \{\neg\ell\}$ ) ) return true;    // backtracking: try  $\ell := \text{false}$   
    else return false;  
}
```



# Observations for DPLL in practice

---

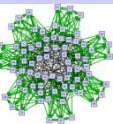
- No deduction of new clauses
  - No dynamic storage allocation
- Algorithm lives (and dies) with unit propagation
  - UP dominates run-time in practice (> 90% UP).
  - This is necessarily so:
    - With 100 variables there are  $2^{100}$  cases without UP
    - With complete UP there are only 99 propagations
    - Typical value in practice: 90 propagations,  $2^{10}$  remaining cases
- Lesson from practice (and secret behind DPLL)
  - Only few decisions are essential, the remaining cases follow as immediate consequences, ruling out many theoretical alternatives.



# Example: SAT-Solving with DPLL

---

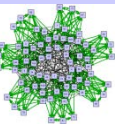
- $S_0 = \{\{x, y, z\}, \{\neg x, y, z\}, \{\neg x\}, \{z, \neg y\}\}$ 
  - unit propagation of  $\neg x$ 
    - $\{\neg x\}$  subsumes  $\{\neg x, y, z\}$ , hence  $\neg x \wedge (\neg x \vee y \vee z) \equiv \neg x$
    - $\{\neg x\}$  unit-resolves with  $\{x, y, z\}$  to  $\{y, z\}$ , and  $\{y, z\}$  subsumes  $\{x, y, z\}$
- $S_1 = \{\{y, z\}, \{z, \neg y\}\}$ 
  - Heuristically choose  $y$  as decision variable:
  - Case 1: let  $y=1$
  - $S_2 = \{\{y\}, \{y, z\}, \{z, \neg y\}\}$
  - unit propagation of  $y$  yields  $S_3 = \{\{z\}\}$ ,
  - unit propagation of  $z$  yields  $S_4 = \{ \}$ , return true.



# Variable Selection Heuristics for DPLL

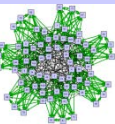
---

- Some Heuristics for variable selection:
  - Choose the literal which occurs most often.
    - Then the formula is simplified in the most places
  - Choose a literal  $L$  from 2-clause (binary clause)  $\{K, \neg L\}$ .
    - Then  $K=1$  if  $L=1$ , resp.  $L=0$  if  $K=0$ , because clause encodes  $(L \rightarrow K)$
    - Clever choice of  $K$  resp.  $L$  immediately leads to UP, eliminating a decision
  - Choose a literal from a short(est) clause.
    - Will soon produce a binary clause, then a UP
  - For each literal  $L$ , compute how  $F$  would be shortened by UP after assigning  $L$ . Choose that  $L$  which has the greatest effect.
    - This simplifies  $F$  most before the next decision.



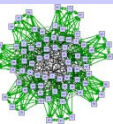
# First Step from DPLL to CDCL: Elementary Learning

- If  $F=0$  after a sequence of variable assignments  $x_i = b_i$ ,  $b_i \in \{0, 1\}$ , i.e.  $\beta(F)=0$ , then we have hit upon a root  $N$  of  $F$ .
  - $N$  is given by a term  $N=(\bigwedge\{x_i\} \wedge \{\neg y_i\})$ ,  $\beta(x_i)=1$ ,  $\beta(y_i)=0$
  - We have  $N \models \neg F$  and hence  $F \models \neg N$
- The negation  $\neg N$  yields a clause  $C$ 
  - $F \models C$  implies  $F \equiv F \wedge C$ , i.e.  $C$  may be safely added to  $F$ .
  - We have „learned“ to avoid this root in the future.
- CDCL: conflict driven clause learning (first idea only)
  - Good:  $C$  need only contain the decision variables in  $N$
  - Bad: in general, many decisions were taken before hitting the root, not all of them relevant for the root.  $C$  is still far too long, does not capture the root condition precisely, i.e.  $C$  is too special and of limited re-use.



# Example: Elementary CDCL

- $S_0 = \{\{x, y\}, \{\neg y, z\}, \{\neg z, x\}\}$ 
  - Heuristically choose decision variable  $x$  :
  - Case 1: let  $x=0$
  - $S_1 = \{\{\neg x\}, \{x, y\}, \{\neg y, z\}, \{\neg z, x\}\}$
  - unit propagation  $\neg x$
  - $S_2 = \{\{y\}, \{\neg y, z\}, \{\neg z\}\}$
  - unit propagation  $y$
  - $S_3 = \{\{z\}, \{\neg z\}\}$
  - unit propagation  $z$
  - $S_4 = \{\{\}\}$ , return false
- We learn that  $S_0 = 0$  if  $x=0$ , hence  $C = \{x\}$ .
  - If we restart  $\text{SAT}(S_0)$ , we are directed away from the root.



# Example: Elementary CDCL

- Analysis of example  $S_0 = \{\{x, y\}, \{\neg y, z\}, \{\neg z, x\}\}$

x	y	z	$S_0$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

- From the previous variable assignments we learned: we hit the root  $N=(x=0, y=1, z=1)$ .
  - Since  $x$  was the only decision variable, we further learn that  $N$  is part of a cluster of 4 roots.
- In the future, the learned clause  $C = \{x\}$  directs us away from any of those 4 roots
    - E.g. after a fresh call and attempted propagations  $\neg z$  und  $\neg y$

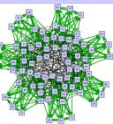




# Principle of Learning in CDCL

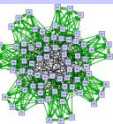
---

- Learning to avoid a bad sequence of decisions
  - So far: we learn to avoid the sequence of decisions with which we hit the root
  - But not all of these decisions may be relevant for the root.
- Key insight: start learning process with *conflict clause*  $K$ 
  - Conflict clause (failure clause)  $K$  is the clause which becomes empty in Step 2 of DPLL, i.e.  $\beta(K)=0$ .
  - The failure is caused by all literals in  $K$  becoming 0. This set is relatively small.
  - Now we need to find only those decisions, which caused these literals to become 0.



# Principle of Learning in CDCL

- Learning process starting with *conflict clause*  $K$ 
  - For conflict clause  $K$ ,  $\beta(K)=0$  is *always* due to a unit propagation in Step 1 (decisions never fail a clause).
  - As a consequence of a decision, some clause  $R$  („reason“-clause) became unit and forced  $L=1$  for a literal  $L$  in  $R$ .
  - As an immediate consequence,  $K$  failed, i.e. the sole remaining literal  $L'$  in unit clause  $K$  was forced to  $L'=0$ .
  - Hence  $L$  und  $L'$  are complementary literals and  $R$  and  $K$  have a resolvent without this literal. Since  $K$  and  $R$  were unit, this resolvent is also a failure clause  $K'$  (under the current assignment). Now iterate the process ....
  - In this way, all UP-literals can be successively removed, until only decision variables (without reason clause) remain.
  - (there are still some open details)



# Example: Principle of Learning in CDCL

- $S_0 = \{\{x, y\}, \{\neg y, z\}, \{\neg z, x\}\}$ . We made the assignments:
- $x=0$  (Decision),  $y=1$  (Unit Propagation),  $z=1$  (Unit Propagation)
  - Now  $\{\{x, y\}, \{\neg y, z\}, \{\neg z, x\}\}$ .
  - Conflict clause is  $K=\{\neg z, x\}$ , Reason for conflict is  $R = \{\neg y, z\}$
  - Resolvent on conflict literal  $z$  (first learnt clause) is  $L_1=\{x, \neg y\}$
  - Notice that  $L_1$  is false under current assignment. It contains both a decision variable  $x$  and a unit propagation variable  $y$ .
  - We can get rid of  $\neg y$  by resolving with *its* reason  $\{x, y\}$ . So  $L_2=\{x\}$
  - Now if we backtrack to before the assignment on  $x$ , there is no decision left:  $x=1$  now becomes a unit propagation.
  - In general we continue learning clauses until we hit the first „UIP clause“ (unique implication point): It contains a single variable on the highest level of assignment. After backtracking it is unit.

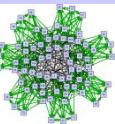


# Example: Principle of Learning in CDCL

- Analysis of Example  $S_0 = \{\{x, y\}, \{\neg y, z\}, \{\neg z, x\}\}$

x	y	z	$S_0$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

- From failure clause  $K = \{\neg z, x\}$  we learn: we hit the *cluster* of roots  $N = (x=0, z=1)$
  - After resolution of  $K$  with  $R_2 = \{\neg y, z\}$  we learn  $\{\neg y, x\}$ , i.e.  $N' = (x=0, y=1)$  is a ``neighboring`` cluster of roots.
  - After resolution with  $R_1 = \{x, y\}$  we learn  $\{x\}$  with cluster  $N'' = (x=0)$ .
- Future hits into cluster  $N'' = (x=0)$  (e.g. by a different UP sequence) are precluded by learned clause  $K' = \{x\}$ .



# Example: CDCL based proof learning

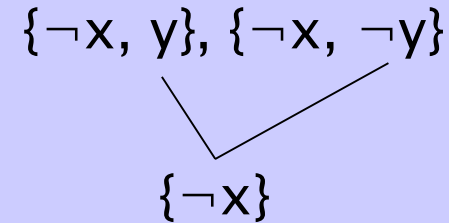
- Clause set  $S_0 = \{ \{x, y\}, \{x, \neg y\}, \{\neg x, y\}, \{\neg x, \neg y\} \}$ 
  - $S_0(x=1) = \{ \{1, y\}, \{1, \neg y\}, \{0, y\}, \{0, \neg y\} \}$  : propagate unit  $y=1$
  - $S_0(x=1, y=1) = \{ \{1, 1\}, \{1, 0\}, \{0, 1\}, \{0, 0\} \} = 0 = \text{conflict!}$
  - A decision ( $x=1$ ) forced conflicting propagations  $y=1$  and  $y=0$ , obviously by 2 clauses containing  $\{..., y, ..\}$  and  $\{..., \neg y, ..\}$
  - Hence there is a resolvent on  $y$ , in this case  $\{\neg x, y\}, \{\neg x, \neg y\} \vdash \{\neg x\}$ .
  - Add  $\{\neg x\}$  to  $C$ , because it is a logical consequence of  $C$ .
  - Backtrack to just before the decision on  $x$  (no matter how far!). Now  $x=0$  is a forced unit propagation by  $\{\neg x\}$  (no more decision)
  - $S_1 = \{ \{\neg x\}, \{x, y\}, \{x, \neg y\}, \{\neg x, y\}, \{\neg x, \neg y\} \}$  : propagate  $x=0$
  - $S_1(x=0) = \{ \{1\}, \{0, y\}, \{0, \neg y\}, \{1, y\}, \{1, \neg y\} \}$  : propagate  $y=1$
  - $S_1(x=0, y=1) = \{ \{1\}, \{0, 1\}, \{0, 0\}, \{1, 1\}, \{1, 0\} \} = \text{conflict!}$
  - Hence there is a resolvent on  $y$ , in this case  $\{x, y\}, \{x, \neg y\} \vdash \{x\}$ , add  $\{x\}$  to  $S_1$
  - Without any decision on  $x$ , we have a final conflict in  $S_2 = \{ \{\neg x\}, \dots, \{x\} \} \vdash \square$



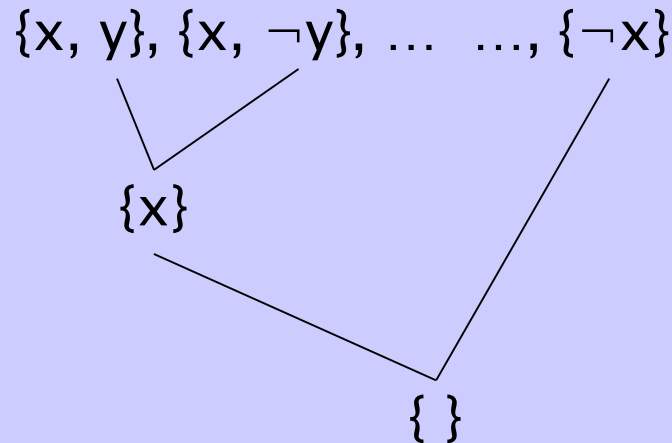
# Resolution proof explaining UNSAT(C)

$$C = \{ \{x, y\}, \{x, \neg y\}, \{\neg x, y\}, \{\neg x, \neg y\} \}$$

Resolution Proof of UNSAT(S(x=1)),  
respectively of  $S \models \{\neg x\}$  :



Final proof of UNSAT(S):

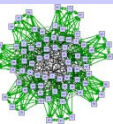


*The answer is easy if you take it logically (Paul Simon)*



# Note on UnSAT Cores

- Usually, only a (small) subset of  $S$  is unsatisfiable
- This is called an **UnSAT Core**
  - $S$  may contain very many different UnSAT cores
  - The clauses used in the final resolution proof of UnSAT produced by the solver contains an UnSAT Core
- An UnSAT Core is minimal if it becomes SAT if any clause is removed (all clauses are necessary)
  - In practice, cores produced by solvers are close to minimal
  - can be easily reduced to minimal by trial and error
- UnSAT Cores are essential to explain UnSAT
  - If  $\text{UnSAT}(S)$  points to defect in  $S$ , explanation is essential
  - Result is useless without explanation



# Note on Implicants and Prime Implicants

- $S_0 = \{\{x, y\}, \{\neg y, z\}, \{\neg z, x\}\}$
- $\text{SAT}(S_0) = \{x=1, z=1, y=1\}$  yields an *implicant*,  $x \wedge y \wedge z \models S_0$ 
  - but  $y$  is unnecessary:  $x \wedge z$  is also an implicant;  $y$  is „don't care“
- A **Prime Implicant** is an implicant which cannot be shortened (minimal implicant)
  - there may be many prime implicants contained in a model
  - different models may/will contain different prime implicants
- Extracting Prime Implicants from the final clause state
  - SAT solver always computes complete models
  - remove any unnecessary literal (all clauses must remain true)
    - Now some literals may become necessary which were not before
  - Now iterate the process as long as possible.





# Summary: Decision Procedures for Propositional Logic

- Davis-Putnam 1960 (*variable elimination rules 2 and 3*)
- DP 1960 (Rule 2): Let  $F$  contain a pure literal  $p$ .
  - Then  $F \equiv (A \vee p) \wedge R$  (where  $p$  not in  $A, R$ )
  - Hence  $F \cong R$  ( $p$  is eliminated from  $F$ )
- DP 1960 (Rule 3) : Let  $F$  contain a complementary literal  $p$ 
  - Then  $F \equiv (A \vee p) \wedge (B \vee \neg p) \wedge R$  (where  $p$  not in  $A, B, R$ )
  - Hence  $F \cong (A \vee B) \wedge R$  ( $p$  is eliminated from  $(A \vee B) \wedge R$ )
  - $F$  increases in size, but is strictly „simpler“ because  $p$  is eliminated.
- Davis-Logemann-Loveland 1962 (*splitting rule*):
  - $F$  is inconsistent *iff* both  $(A \wedge R)$  and  $(B \wedge R)$  are inconsistent
    - No fresh CNF conversion necessary, split in 2 smaller formulas, divide&conquer
- SAT-Solving (optimization of *splitting rule*):
  - $F$  inconsistent iff both  $F|_{p=0}$  and  $F|_{p=1}$  inconsistent
- Resolution:  $(A \vee p) \wedge (B \vee \neg p) \wedge R \equiv (A \vee p) \wedge (B \vee \neg p) \wedge (A \vee B) \wedge R$



# Literature

---

1. M. Davis and H. Putnam. A computing procedure for quantification theory. *J.ACM* 7(3), 1960
2. M. Davis, G. Logemann and D. Loveland. A machine program for theorem proving. *C.ACM* 5, 1962
3. A. Biere, M. Heule, H. van Maaren, T. Walsh (eds.). *Handbok of Satisfiability*. IOS Press 2009. (Comprehensive current account of SAT based methods)
4. M. Davis and H. Putnam. A computing procedure for quantification theory. *J.ACM* 7(3), 1960
5. J. P. Marques-Silva. *Search Algorithms for Satisfiability Problems in Combinatorial Switching Circuits*. PhD Thesis, U. Michigan, 1995
6. J. P. Marques-Silva, K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In: *Intl. Conf. Computer Aided Design.*, Nov 1996.
7. J. P. Marques-Silva, K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. In: *IEEE Transactions on Computers.*, May 1999.
8. D. E. Knuth. Satisfiability. *The Art of Computer Programming Vol 4 Fasc. 6*. Addison Wesley, 2016

