Normal Forms for Boolean Expressions

A ***NORMAL FORM*** defines a class expressions s.t.

a. Satisfy certain structural properties
b. Are usually universal: able to express every boolean function

1. Disjunctive Normal Form (DNF)
   - Sum Of Products of *literals*, i.e., a variable or its negation

Example: xy'z + yz + w

2. Conjunctive Normal Form (CNF)
   - Product of CLAUSES, i.e., sum of literals

Example: (z+w).(x+y+z'+w), (x+y'+z).(y+z).w'

3. Negation Normal Form (NNF): Negation appears only at leav

Example: (x+yz).y'   Counter Example: (a'.b)'+c'

Satisfiability/validity of DNF and CNF

Satisfiability of formulas in DNF can be checked in linear time.

> A formula in DNF is satisfiable iff at least one of its conjunctions is satisfiable. A conjunction is satisfiable iff for every atomic formula A the conjunction does not contain both A and $\neg A$ as literals.

Satisfiable: $(\neg B \wedge A \wedge B) \vee (\neg A \wedge C)$

Unsatisfiable: $(A \wedge \neg A \wedge B) \vee (C \wedge \neg C)$

Satisfiability/validity of DNF and CNF

Validity of formulas in CNF can be checked in linear time.

A formula in CNF is valid iff all its disjunctions are valid. A disjunction is valid iff for some atomic formula A the disjunction contains both A and $\neg A$ as literals (or the disjunction is empty.)

Valid: $(A \vee \neg A \vee B) \wedge (C \vee \neg C)$

Not valid: $(A \vee \neg A) \wedge (\neg A \vee C)$

Propositional Logic Decidability Complexity

Theorem: *Satisfiability of CNF formulas is NP-complete*

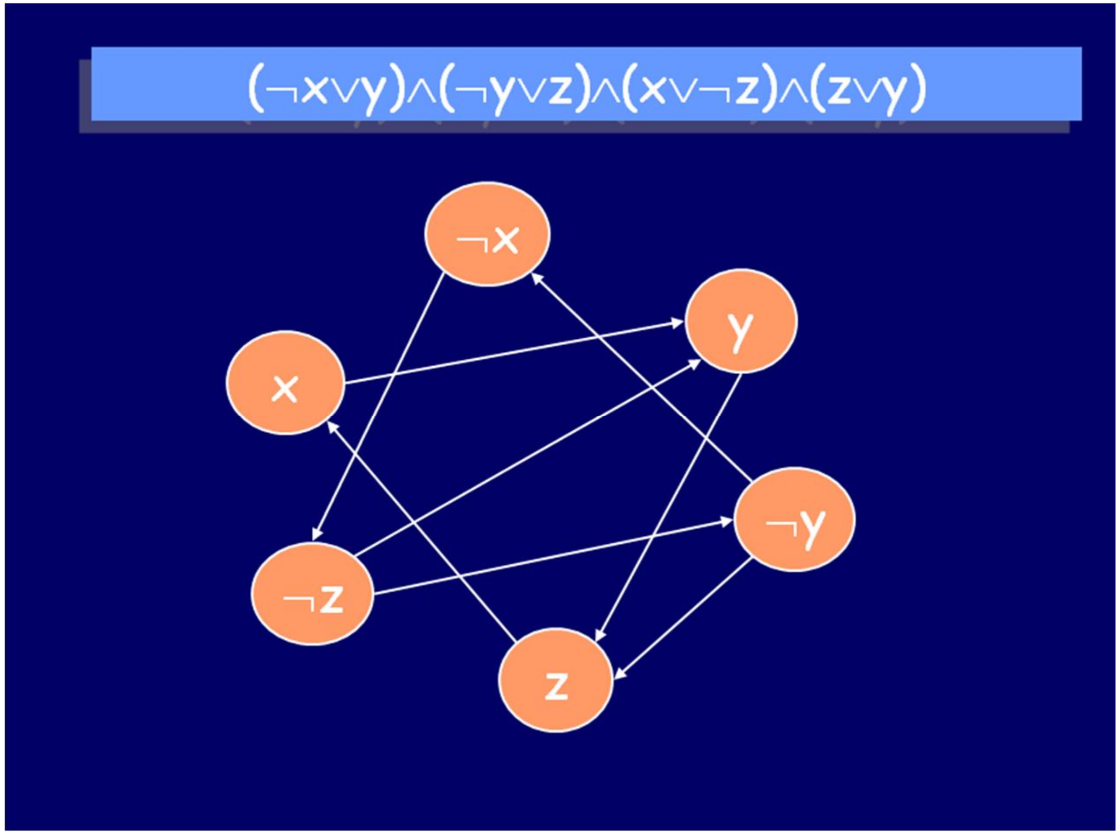
Theorem: *Validity of DNF formulas is NP-complete*

Theorem: *Satisfiability and Validity of arbitrary boolean formulas is NP-complete*

Intuition behind NP-completeness:

Transformation b/w normal forms can have exponential blow-u

2SAT Satisfiability is Polynomial Time



Implication Graph Notes:

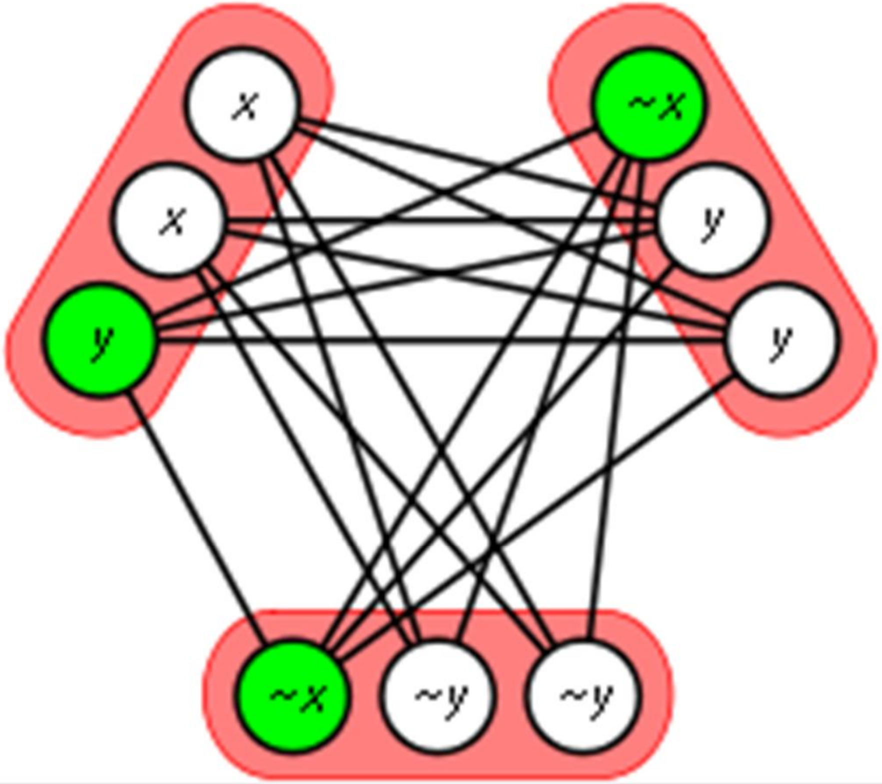
1. Each clause is an implicatior
   e.g., x'+y = x → y
2. Vertex for each literal in claus
3. One edge for each implicatioı

For each variable

Check if there is a path from X to X' as well as from X' to X

Path checking on graph is Poly!!

Reduction of 3SAT CNF to Clique Problem on Grap



The 3-SAT instance $(x \vee x \vee y) \wedge (\neg x \vee \neg y \vee \neg y) \wedge (\neg x \vee y \vee y)$ reduced to a Clique problem. The green vertices form a 3-clique and correspond to the satisfying assignment x=FALSE, y=TRUE.

Theorem:

3SAT and above is NP-comple

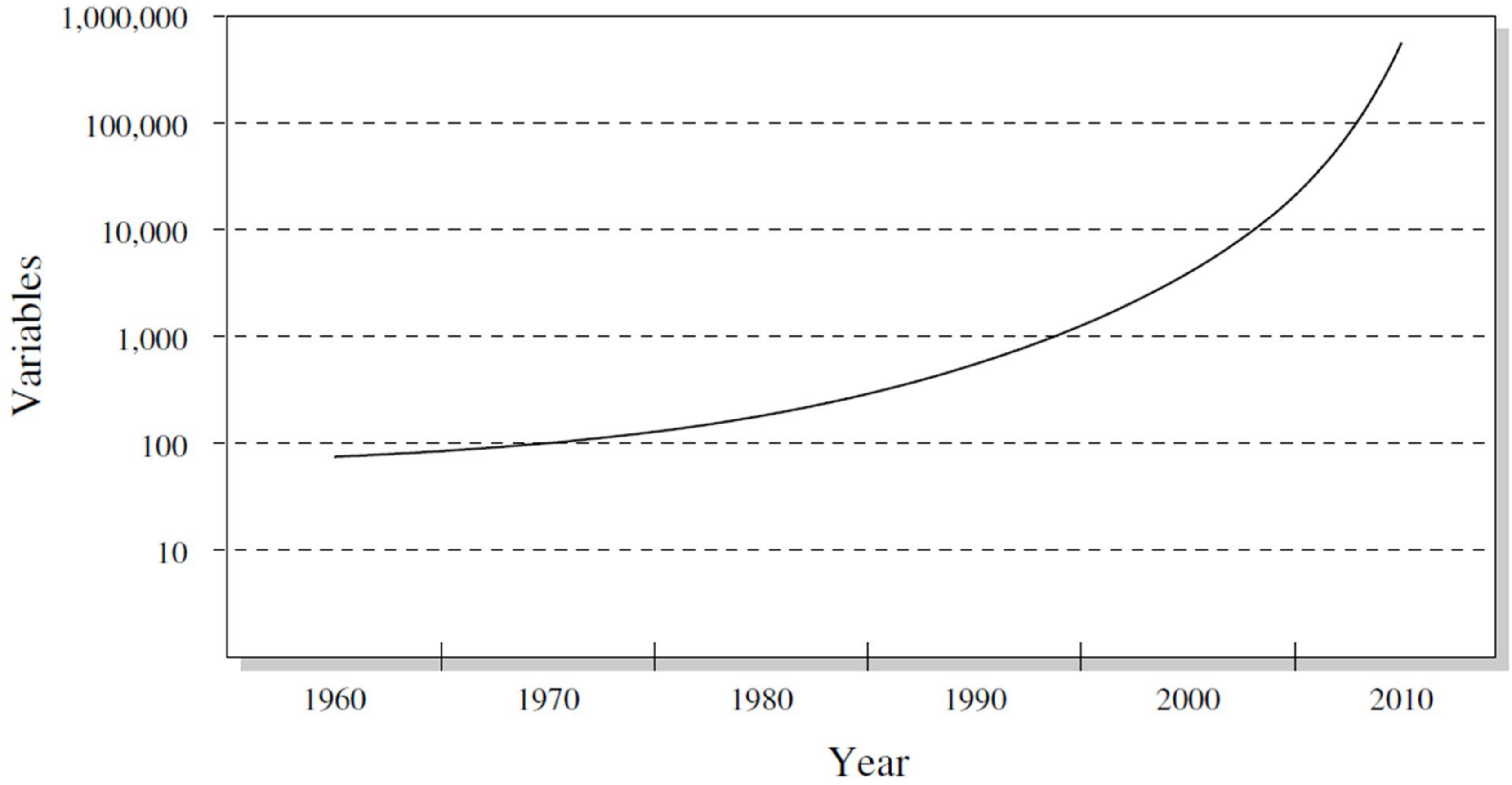
Note: Clique is NP-comple

Are we doomed then?

- No, there are efficient methods that work VERY well for large classes of formulas

- We study two techniques that are the basis for widely used tools in practice
  - **ROBDD**: A compact ***cannonical*** form for arbitrary boolean functions
  - **SAT solving**: An efficient heuristic-based algorithm to check satisfiablity of CNF formulas

SAT Solver Handling Capacity Progress

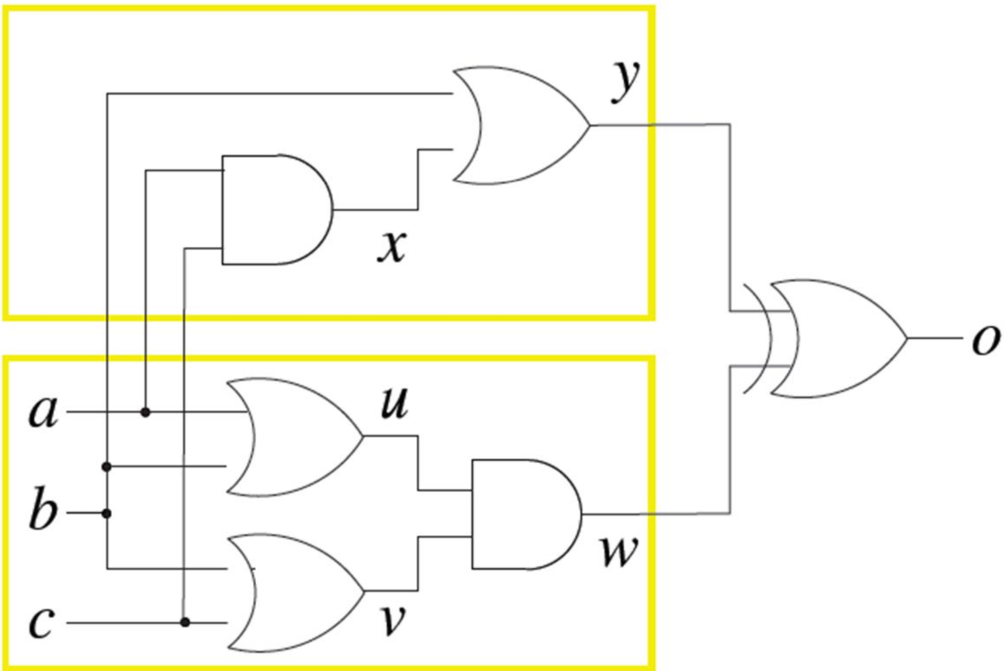


Techniques underlying state-of-art SAT Solvers

- Motivation for SAT
  - BDD is an overkill, especially if just want SAT (e.g., you don't want to do equivalence checking)
  - BDDs often explode without good ordering
- Revolutionary heuristic-based improvements on CNF-based resolution/sat methods
  - Isn't conversion to CNF itself a problem??
- Tseitin Transformation:
  - Can be done with linear increase in size
    - provided you also allow for linear increase in variables

Acknowledgements: Sharad Malik, Princeton, Daniel Kroening, Oxford University

Tseitin Transformation: Circuit to CNF



$$\begin{array}{rcll} & & o & \wedge \\ (x & \leftrightarrow & a \wedge c) & \wedge \\ (y & \leftrightarrow & b \vee x) & \wedge \\ (u & \leftrightarrow & a \vee b) & \wedge \\ (v & \leftrightarrow & b \vee c) & \wedge \\ (w & \leftrightarrow & u \wedge v) & \wedge \\ (o & \leftrightarrow & y \oplus w) & \end{array}$$

$$o \wedge (x \rightarrow a) \wedge (x \rightarrow c) \wedge (x \leftarrow a \wedge c) \wedge \ \ldots$$

$$o \wedge (\overline{x} \vee a) \wedge (\overline{x} \vee c) \wedge (x \vee \overline{a} \vee \overline{c}) \wedge \ \ldots$$

Algorithmic Description of Tseitin Transformation

Tseitin Transformation

1. For each non-input signal s: generate a new variable x_s
2. For each gate: produce input / output constraints as clauses
3. Collect all constraints in a big conjunction

Algorithmic Description of Tseitin Transformation

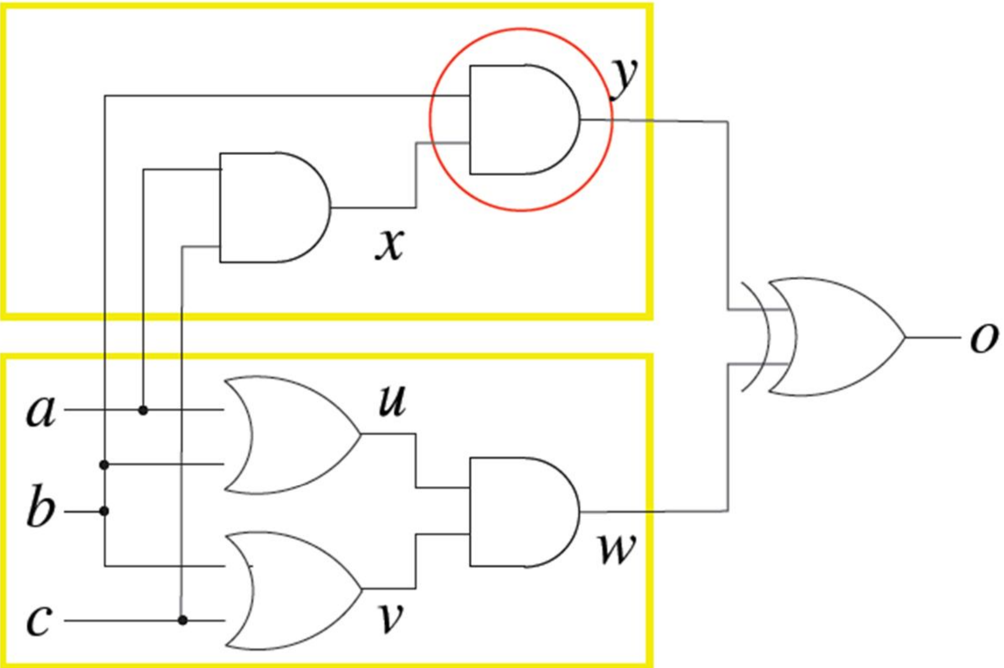
- The transformation is satisfiability-preserving:
  the result is satisfiable iff and only the original formula is satisfiable

- You an get a satisfying assignment for original formula by projecting the satisfying assignment onto the original variables

- Not equivalent in the classical sense to original formula:
  it has new variables

Example SAT: Circuit Equivalence

Let's change the circuit!



$$
\begin{array}{r}
o \;\wedge \\
(x \leftrightarrow a \wedge c) \;\wedge \\
(y \leftrightarrow b \wedge x) \;\wedge \\
(u \leftrightarrow a \vee b) \;\wedge \\
(v \leftrightarrow b \vee c) \;\wedge \\
(w \leftrightarrow u \wedge v) \;\wedge \\
(o \leftrightarrow y \oplus w)
\end{array}
$$

Is the CNF satisfiable?

Some Easy Situations for CNF SAT

- Every literal occurs with the same polarity

  e.g., (a+b')(c'+d)

- Every clause has at least one literal that occurs with same polarity everywhere

  e.g., (a+b')(b+c')

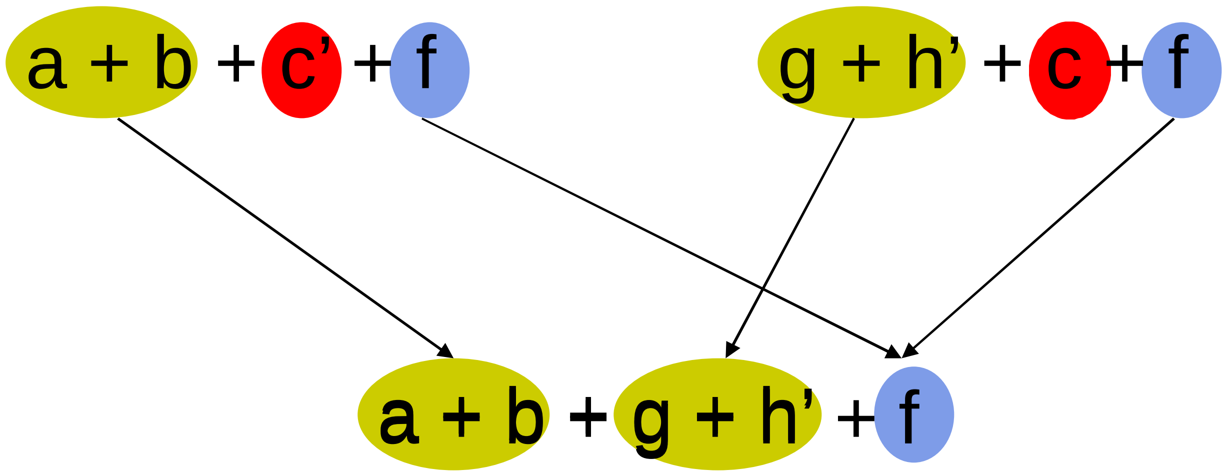
- Nontrivial cases: Every clause has at least one literal that occurs with both polarity everywhere

  e.g., (a+b')(c+d)(b+c'+a')d'

Resolution Rule

$$\frac{\{\varphi_1, \ldots, \chi, \ldots, \varphi_m\} \quad \{\psi_1, \ldots, \neg\chi, \ldots, \psi_n\}}{\{\varphi_1, \ldots, \varphi_m, \psi_1, \ldots, \psi_n\}}$$

- Resolution of a pair of clauses with incompatible variables
- Pick EXACTLY one such pivot variable
- ***Resolvent***, is union of rets of literals in the premise clauses



Soundness: Resolvent EQUISAT Premise CNF

a. Resolvent is true whenever premise CNF is true
i.e., Resolvent is SAT iff premise CNF is SAT

b. If premise CNF is UNSAT Resolvent is UNSAT
e.g., {a} {a'} --> resolvent is empty

Completeness:

It is complete or checking SAT/UNSAT, given a set of clauses

The Timeline

1960: Davis Putnam

Resolution Based

$\approx$10 variables

Davis Putnam Algorithm

M .Davis, H. Putnam, “A computing procedure for quantification theory", *J. of ACM*, Vol. 7, pp. 201-214, 1960

Existential abstraction using resolution

- Iteratively select a variable for resolution till no more variables are left.

$F = (a + b + c)(b + c' + f')(b' + e)$

$\exists b\ F = (a + c + e)(c' + e + f)$

$\exists bc\ F = (a + e + f)$

$\exists bcaef\ F = 1$

SAT

$F = (a + b)(a + b')(a' + c)(a' + c')$

$\exists b\ F = (a)(a' + c)(a' + c')$

$\exists ba\ F = (c)(c')$

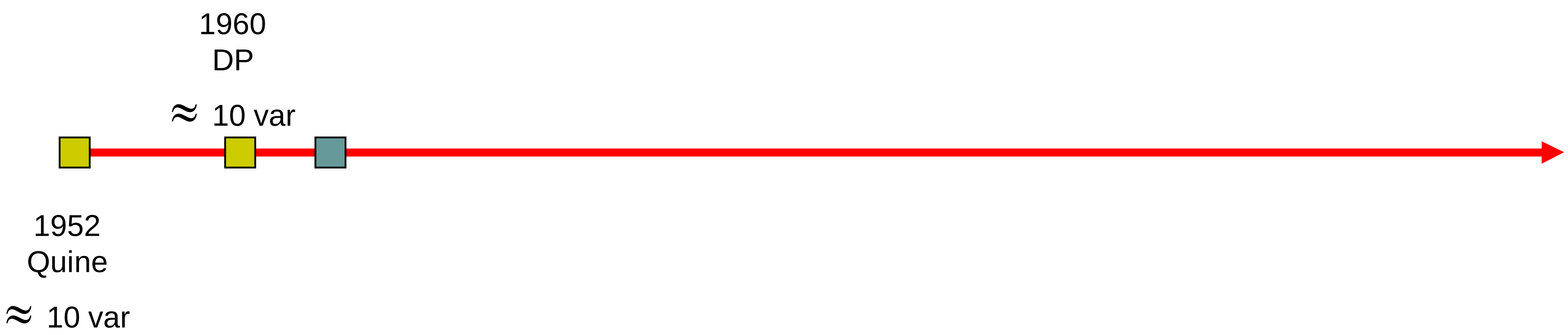
$\exists bac\ F = (\ )$

UNSAT

Potential memory explosion problem!

The Timeline

1962
Davis Logemann Loveland
Depth First Search
≈ 10 var



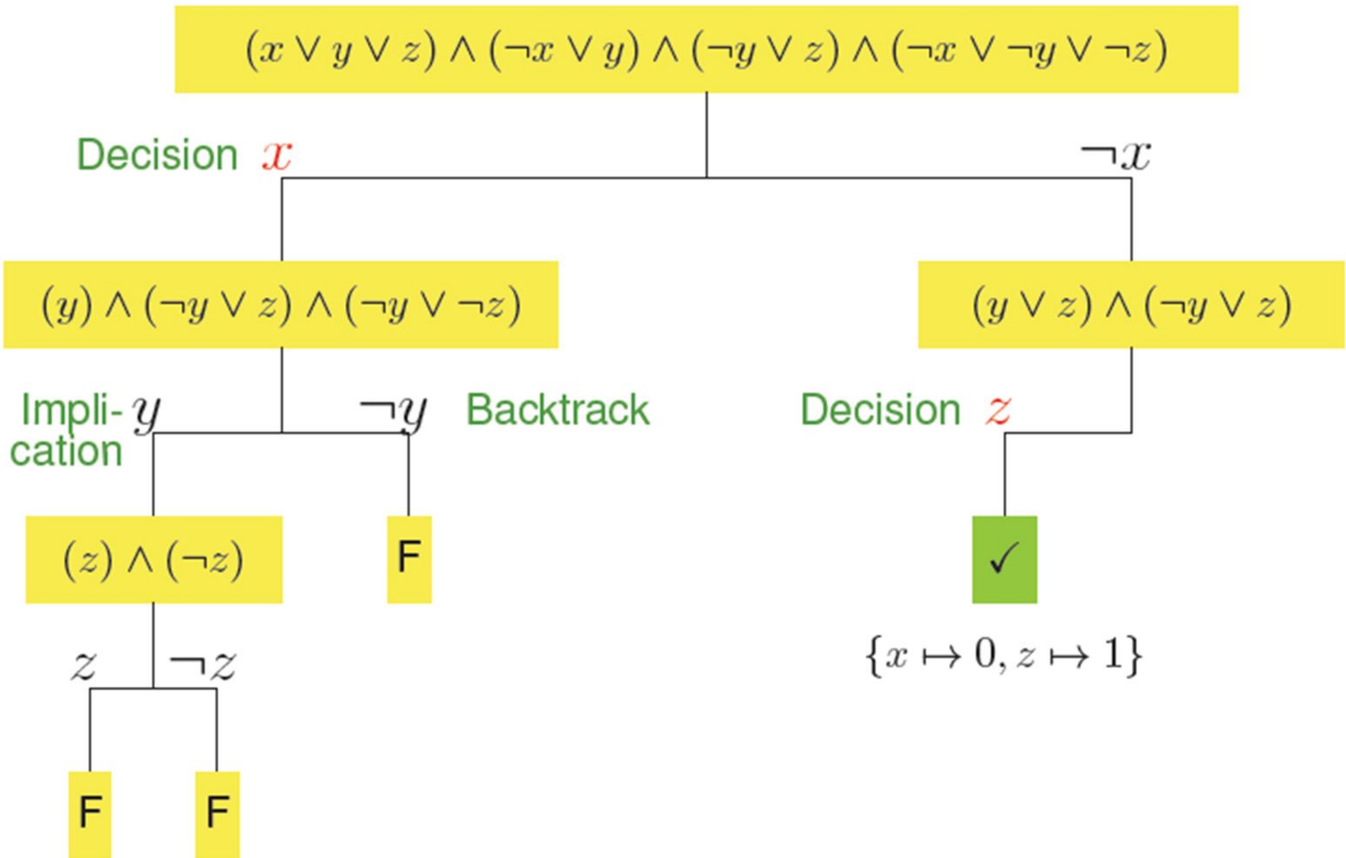
DLL Algorithm

- Davis, Logemann and Loveland

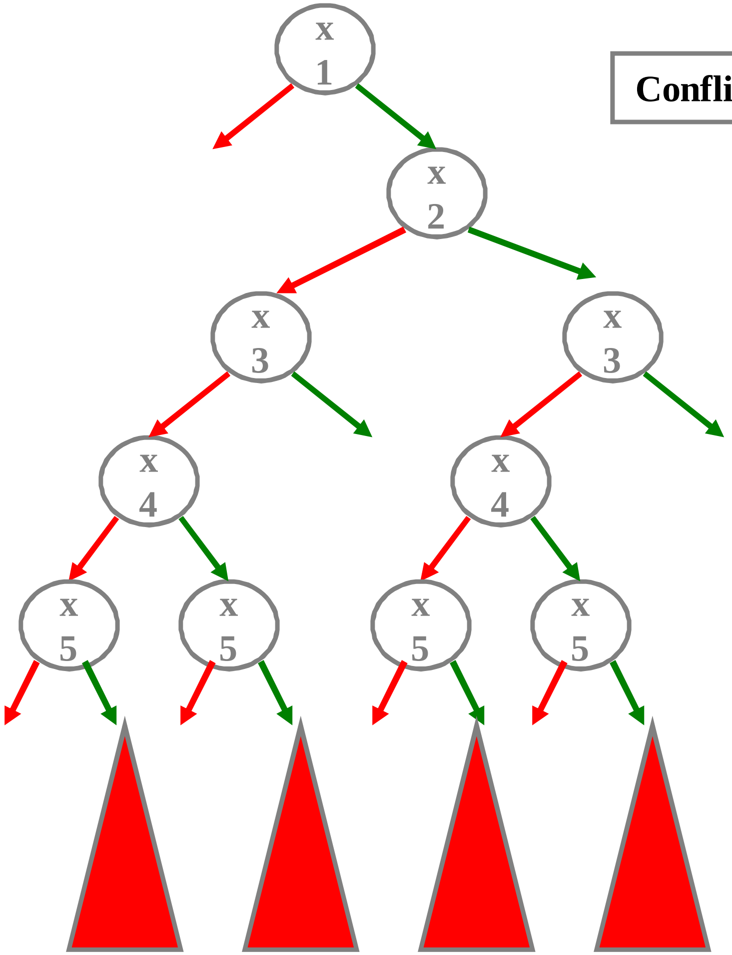
  M. Davis, G. Logemann and D. Loveland, “A Machine Program for Theorem-Proving", *Communications of ACM*, Vol. 5, No. 7, pp. 394-397, 1962
- Also known as DPLL for historical reasons
- Basic framework for many modern SAT solvers

Binary Search

Formula:



What's the big deal?



Conflict clause: x1'+x3+x5'

Significantly prune the search space – learned clause is useful forever!

Useful in generating future conflict clauses.

Notation

Given the partial assignment

$$\{x_1 \mapsto 1,\ x_2 \mapsto 0,\ x_4 \mapsto 1\}$$

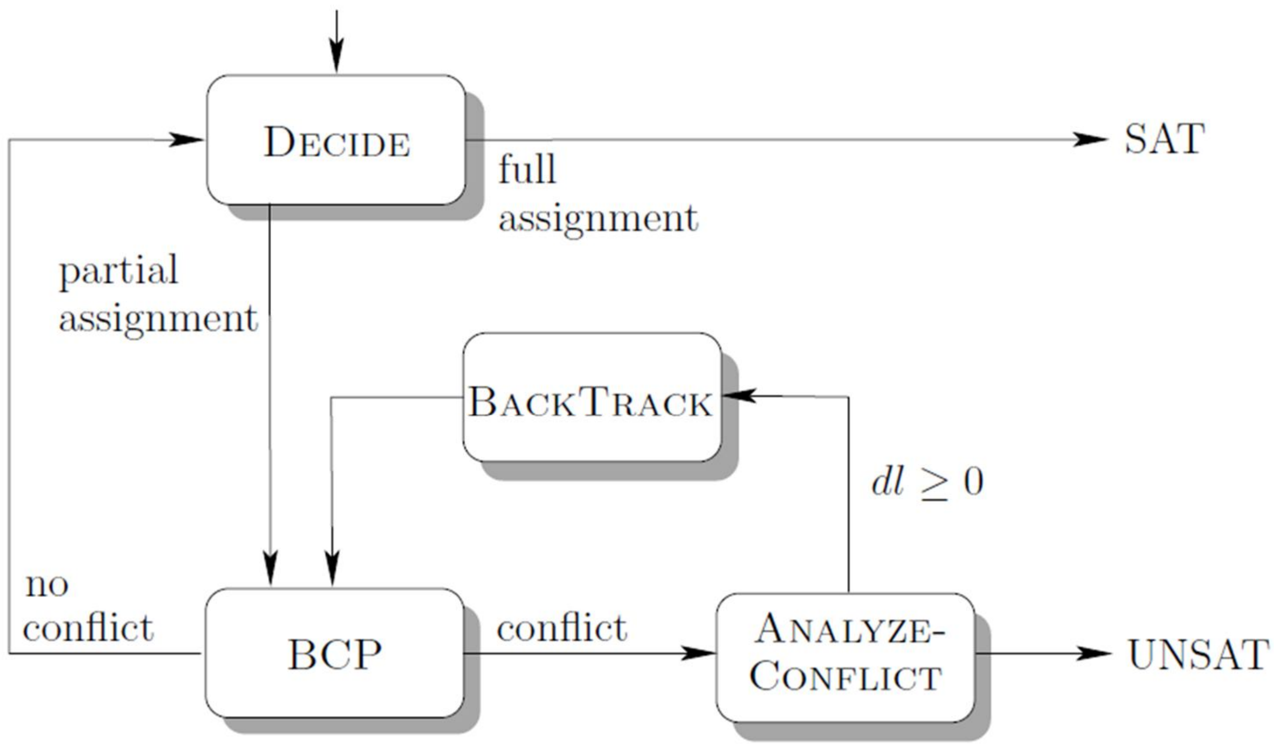
$(x_1 \vee x_3 \vee \neg x_4)$	is satisfied
$(\neg x_1 \vee x_2)$	is conflicting
$(\neg x_1 \vee \neg x_4 \vee x_3)$	is unit
$(\neg x_1 \vee x_3 \vee x_5)$	is unresolved.

Basic DPLL

Given the partial assignment

$$\{x_1 \mapsto 1,\ x_2 \mapsto 0,\ x_4 \mapsto 1\}$$

$(x_1 \vee x_3 \vee \neg x_4)$ is satisfied
$(\neg x_1 \vee x_2)$ is conflicting
$(\neg x_1 \vee \neg x_4 \vee x_3)$ is unit
$(\neg x_1 \vee x_3 \vee x_5)$ is unresolved.



Basic DPLL

```
function DPLL
    if BCP() = 'conflict' then return 'Unsatisfiable';
    while (TRUE) do
        if ¬DECIDE() then  return 'Satisfiable';
        else
            while (BCP() = 'conflict') do
                backtrack-level := ANALYZE-CONFLICT();
                if backtrack-level < 0 then
                    return 'Unsatisfiable';
                else
                    BACKTRACK(backtrack-level);
```

- DECIDE: Choose next variable and value
- BCP: Propagate implications of unit clauses
- ANALYZE-CONFLICT: Determine backtracking level

Implications and Boolean Constraint Propagation

- Implication
  - A variable is forced to be assigned to be True or False based on previous assignments.
- Unit clause rule (rule for elimination of one literal clauses)
  - An unsatisfied clause is a unit clause if it has exactly one unassigned literal.

Satisfied Literal

Unsatisfied Literal

Unassigned Literal

$$(a + b' + c)(b + c')(a' + c')$$

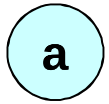
a = T, b = T, c is unassigned

  - The unassigned literal is implied because of the unit clause.
- Boolean Constraint Propagation (BCP)
  - Iteratively apply the unit clause rule until there is no unit clause available.
  - a.k.a. Unit Propagation
- Workhorse of DLL based algorithms.

Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

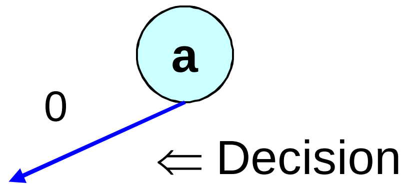
Basic DPLL Procedure - DFS



(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)

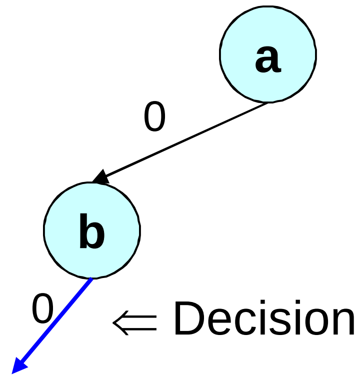
Basic DPLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



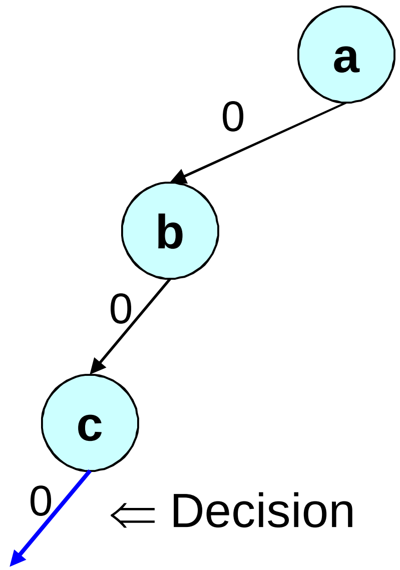
Basic DPLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



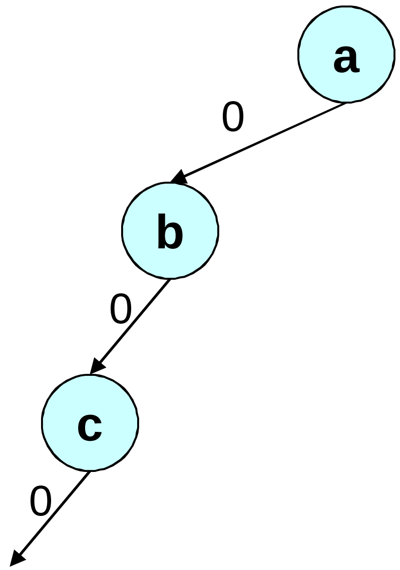
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

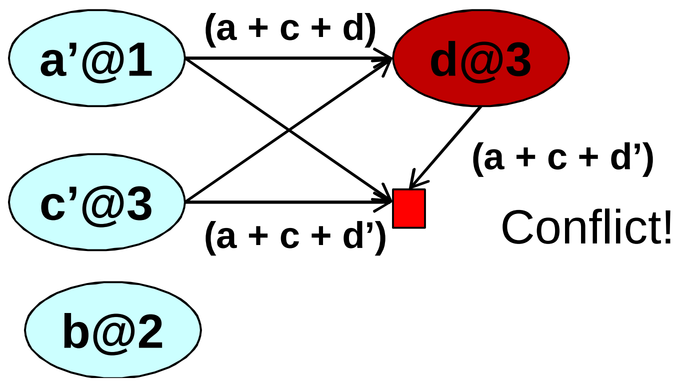


Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

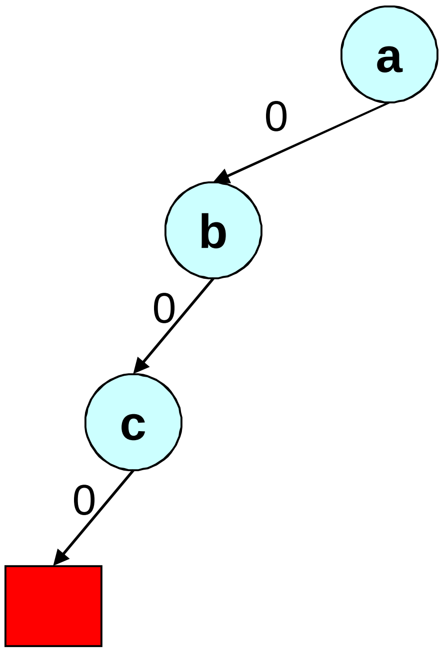


Implication Graph

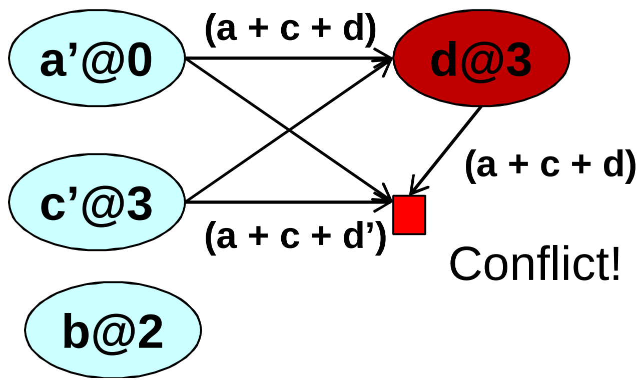


Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

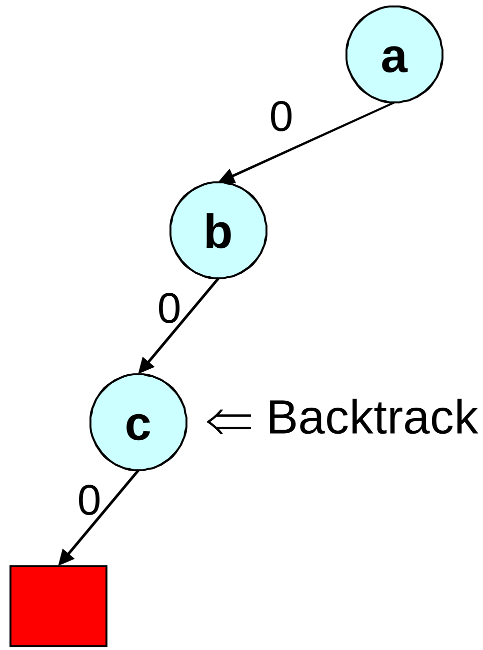


Implication Graph



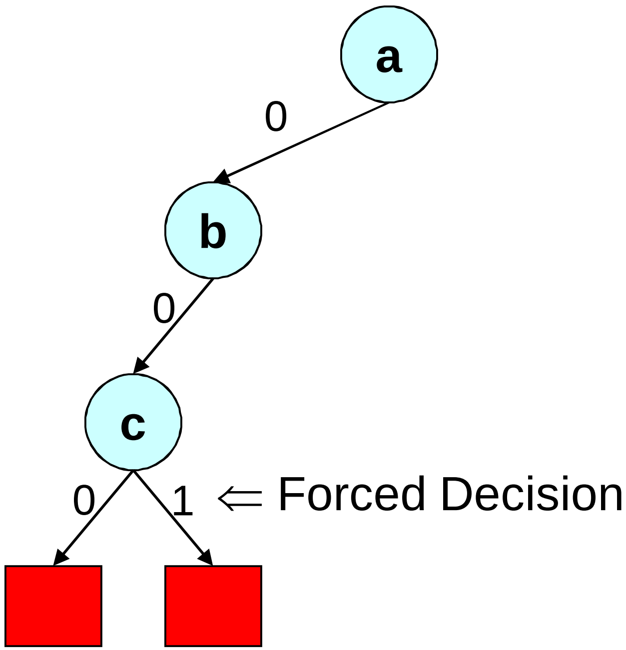
Basic DPLL Procedure - DFS

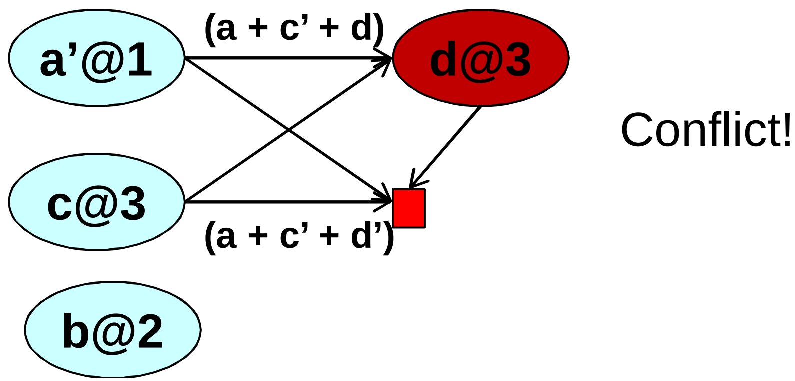
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

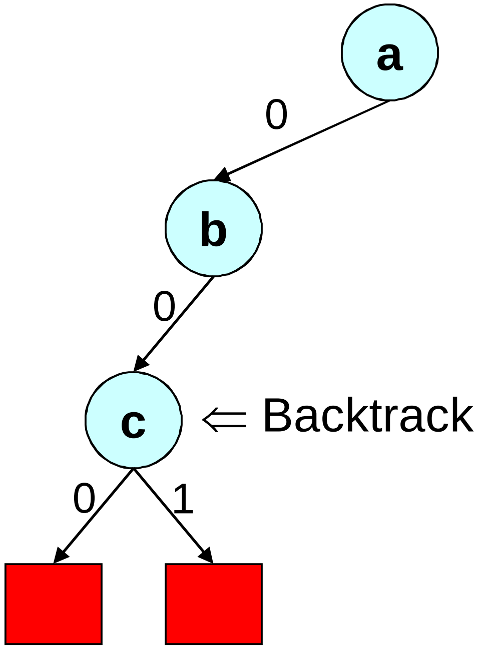
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)





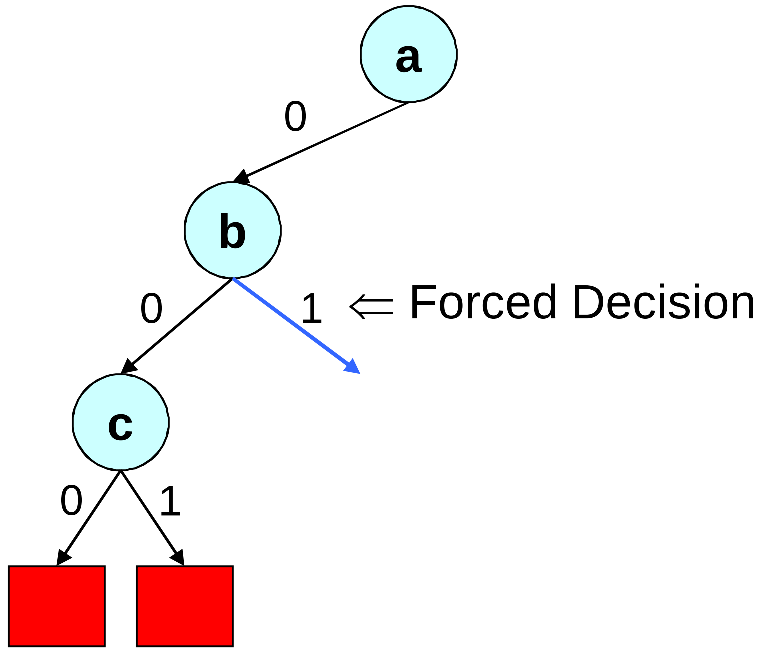
Basic DPLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



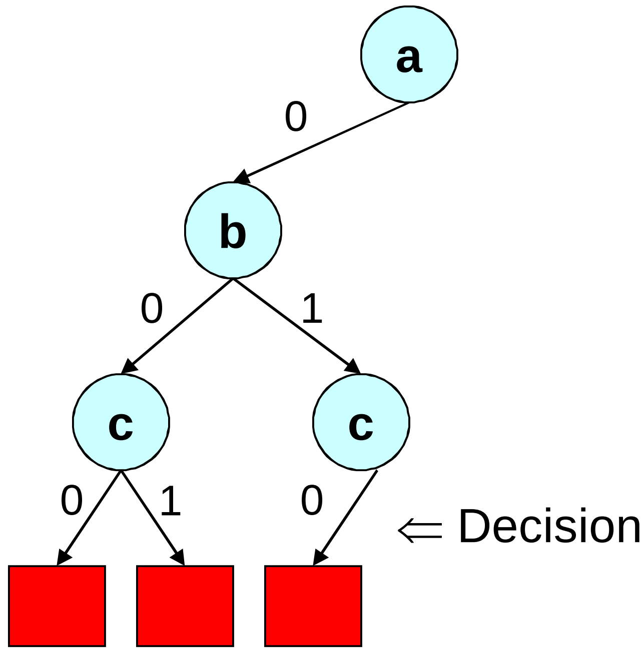
Basic DPLL Procedure - DFS

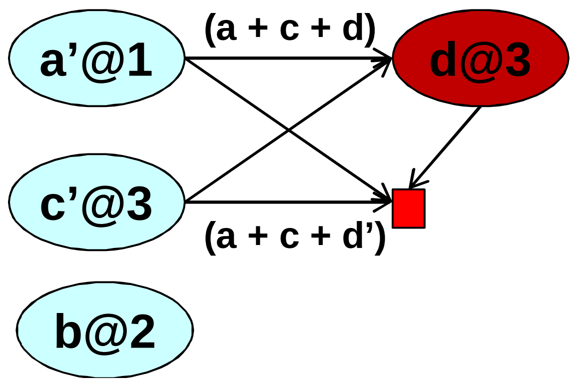
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

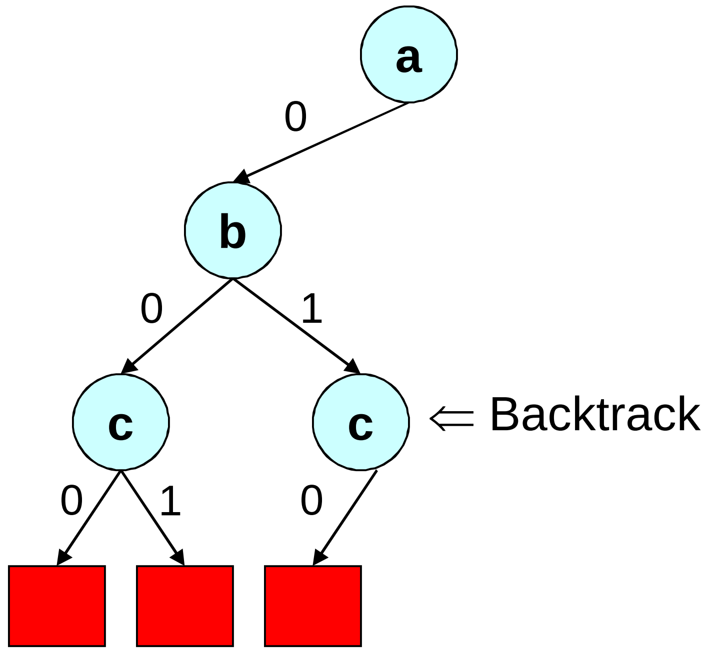




Conflict!

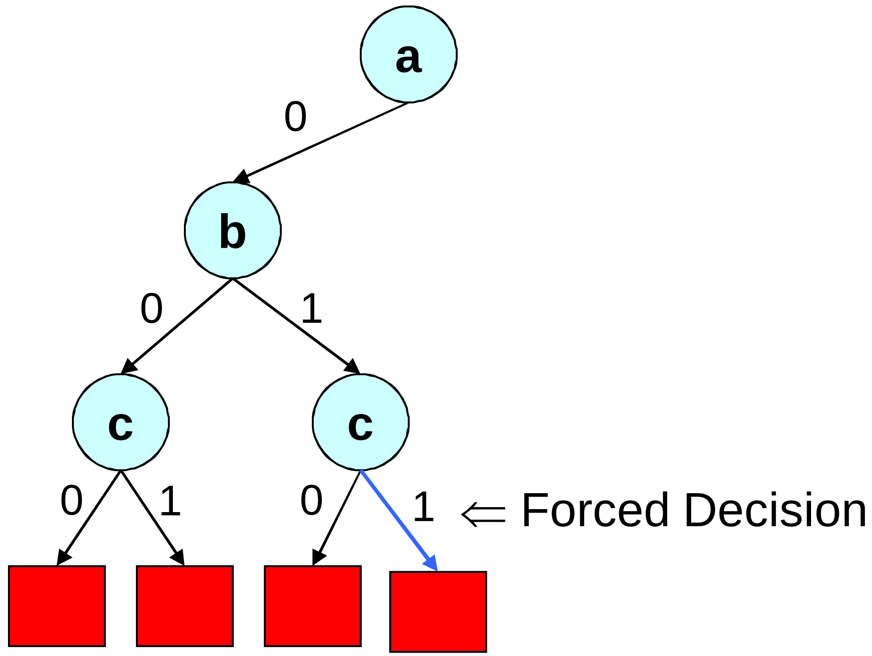
Basic DPLL Procedure - DFS

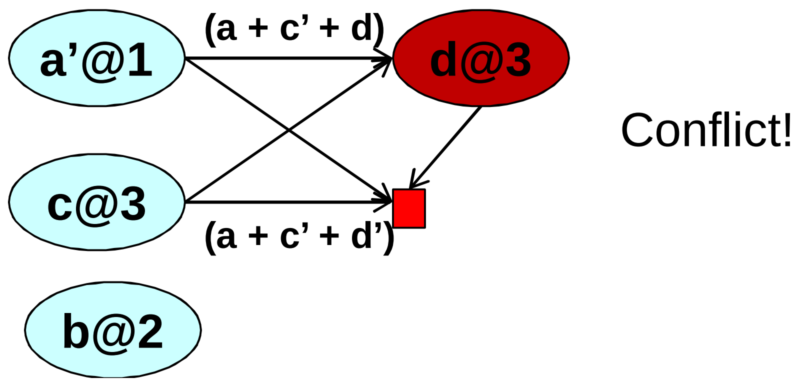
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

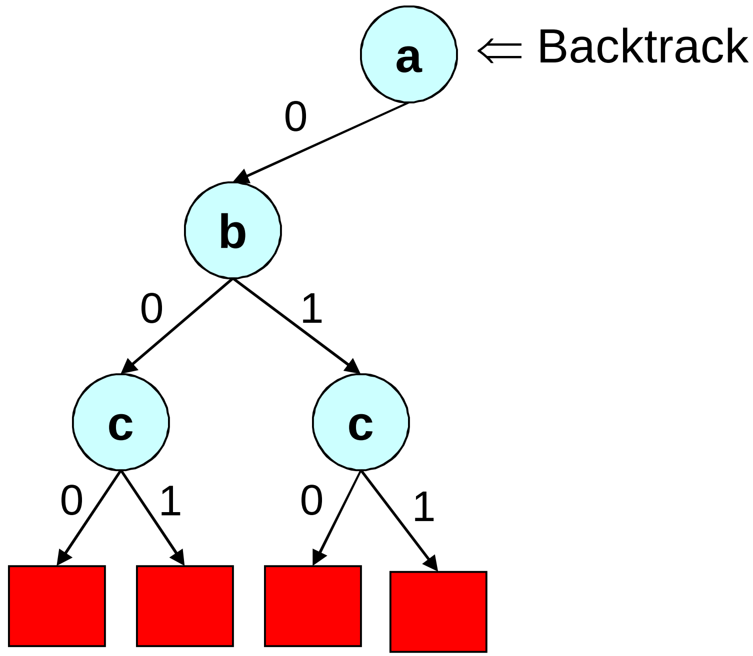
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)





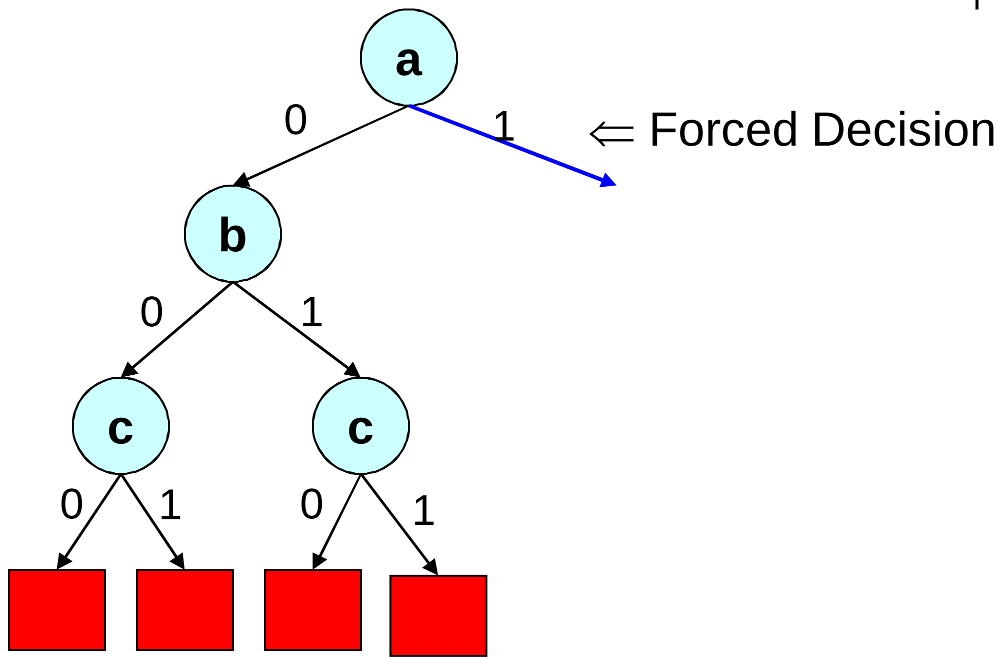
Basic PProcedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



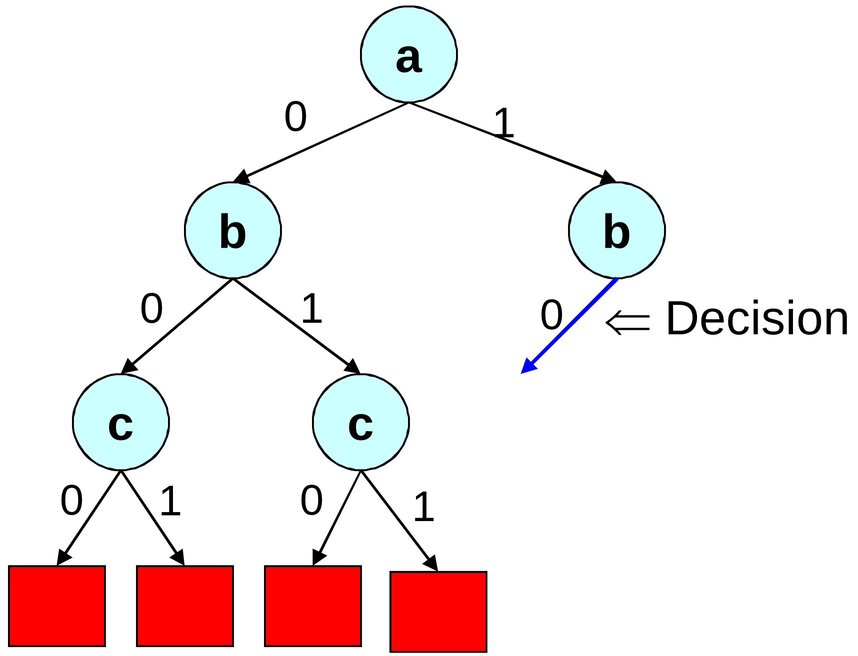
Basic DPLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



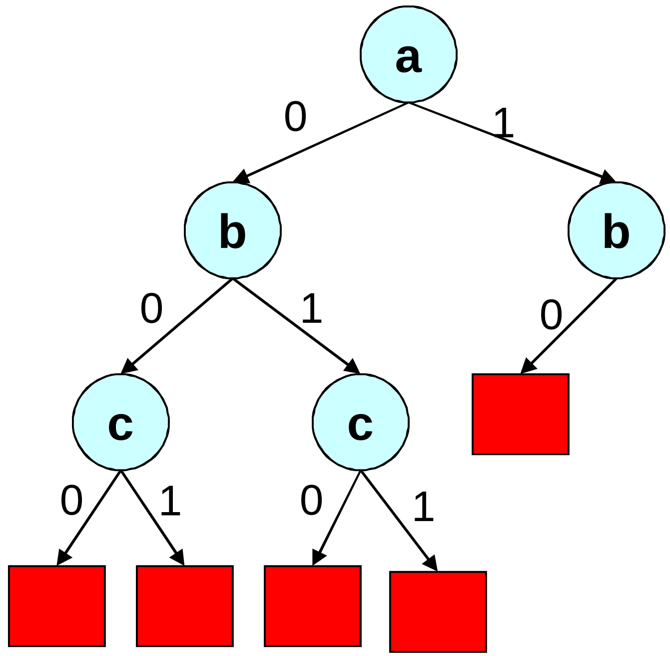
Basic DPLL Procedure - DFS

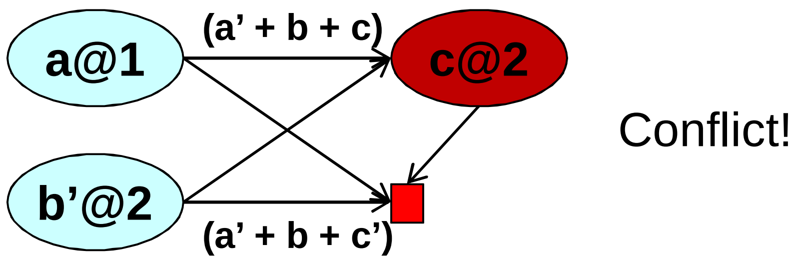
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



Basic DLL Procedure - DFS

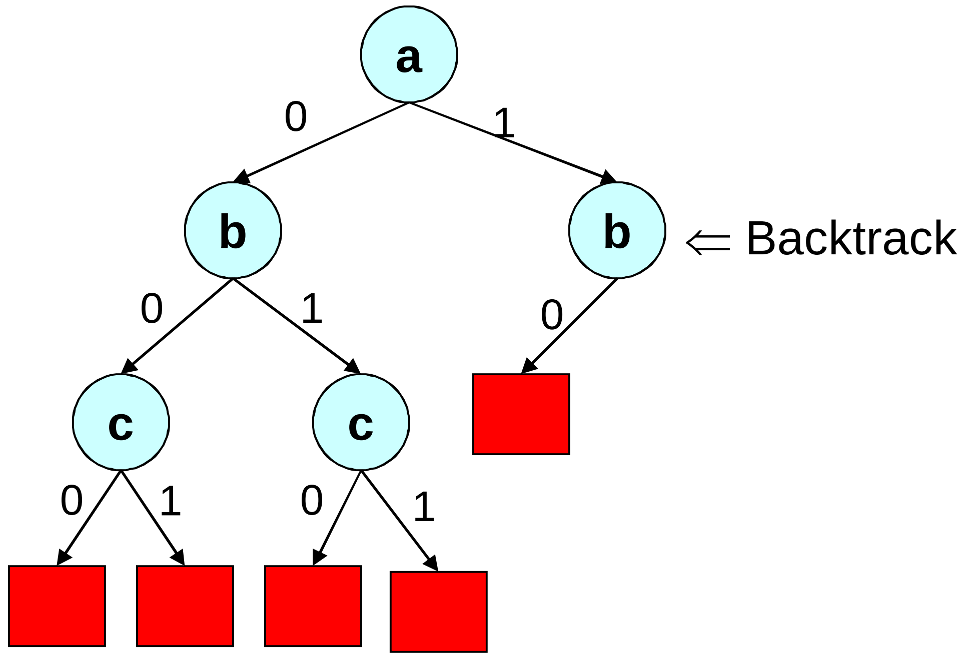
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)





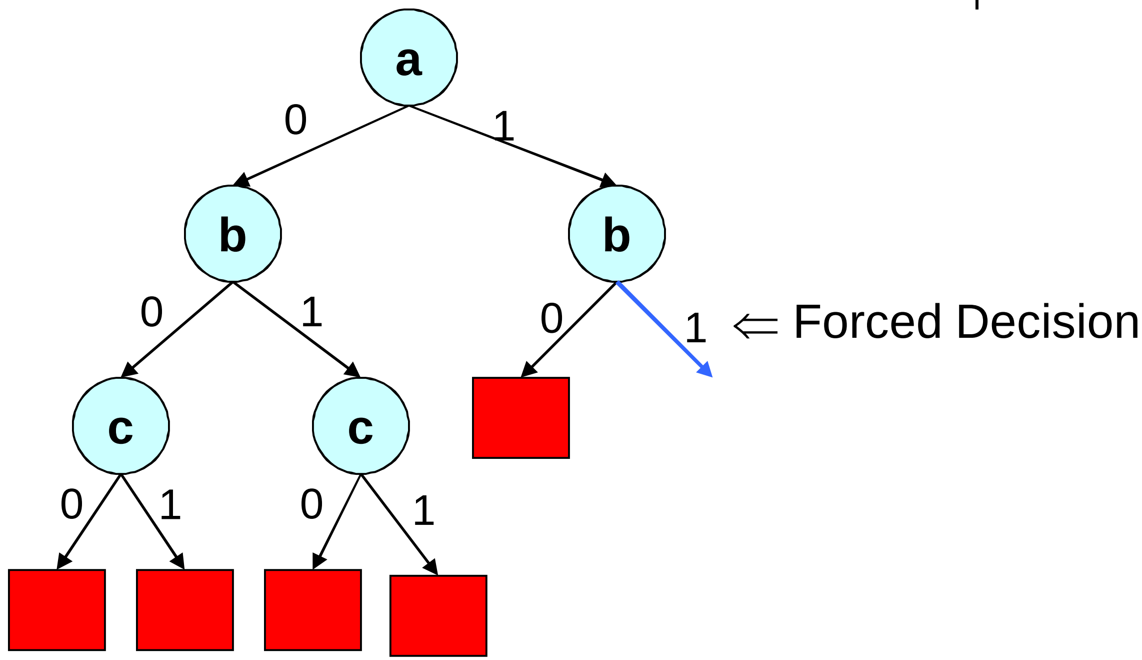
Basic DPLL Procedure - DFS

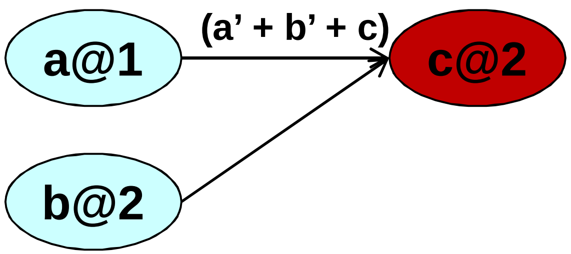
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

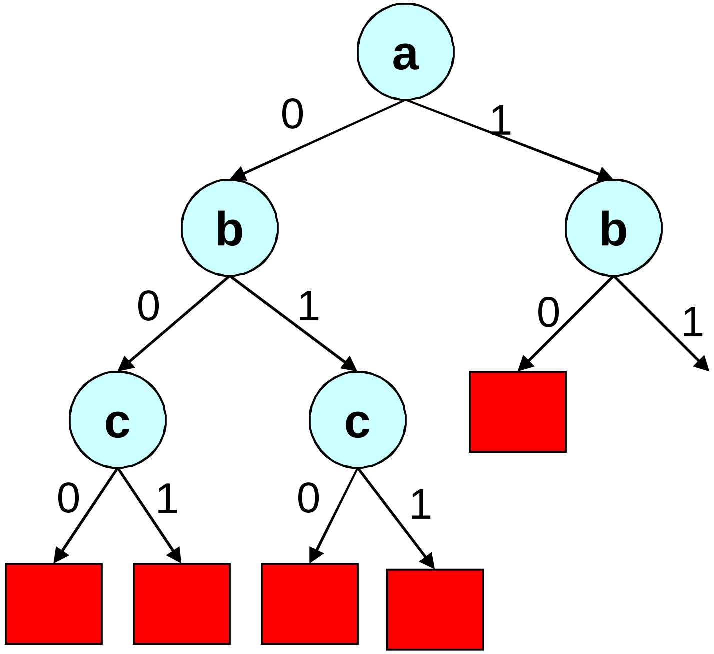
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)

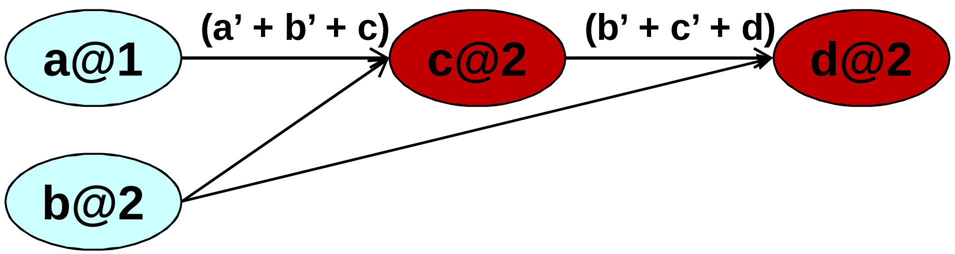




Basic DLL Procedure - DFS

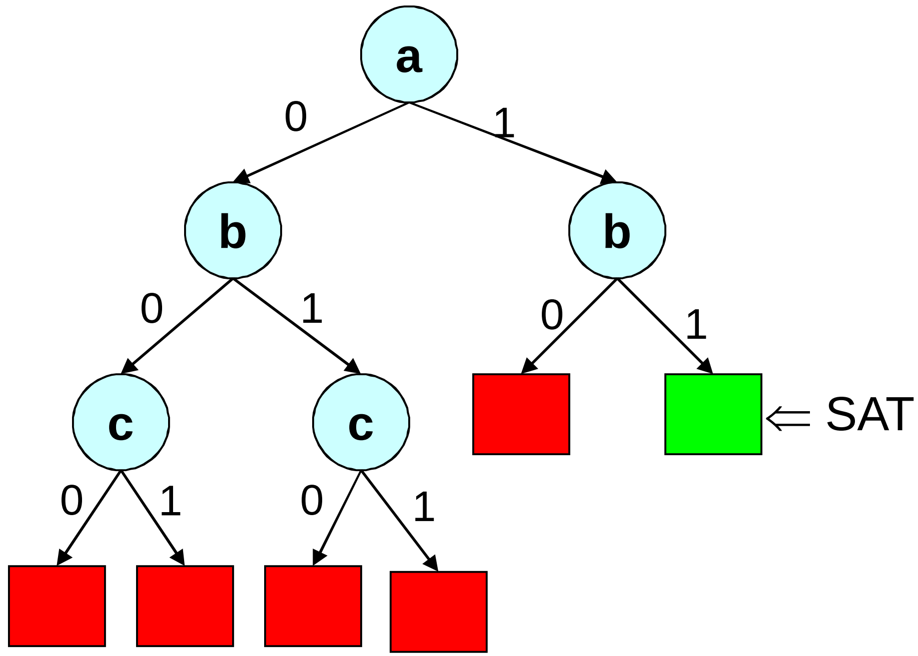
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

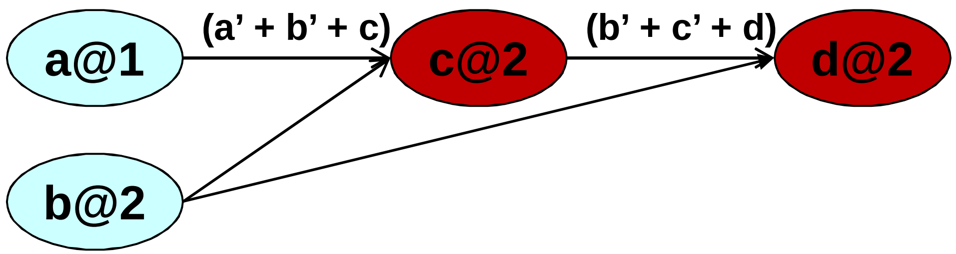




Basic DLL Procedure - DFS

(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



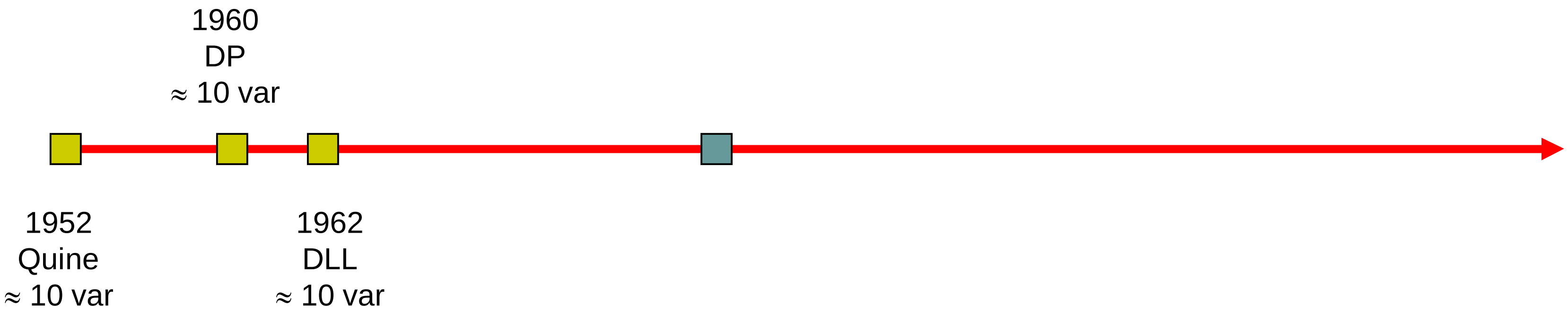


Features of DPLL

- Eliminates the exponential memory requirements of DP
- Exponential time is still a problem
- Limited practical applicability – largest use seen in automatic theorem proving
- Very limited size of problems are allowed
  - 32K word memory
  - Problem size limited by total size of clauses (1300 clauses)

The Timeline

1986
Binary Decision Diagrams (BDDs)
≈100 var

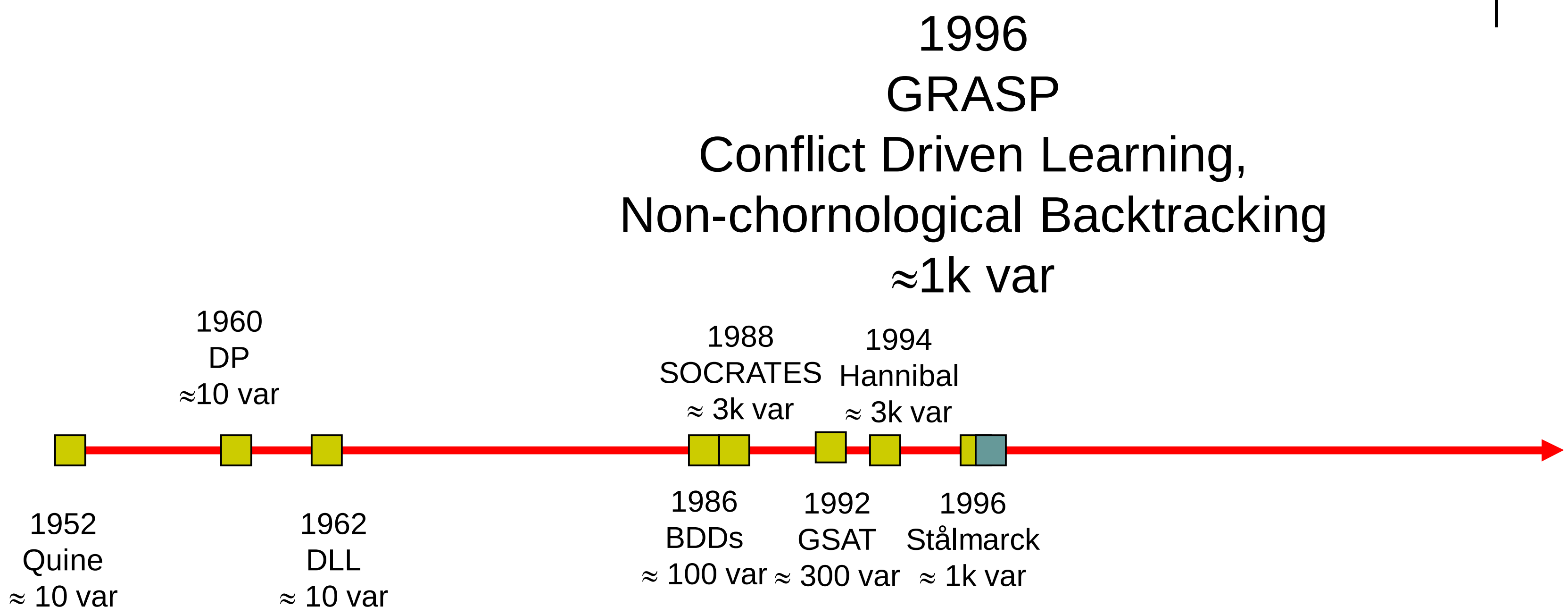


Using BDDs to Solve SAT

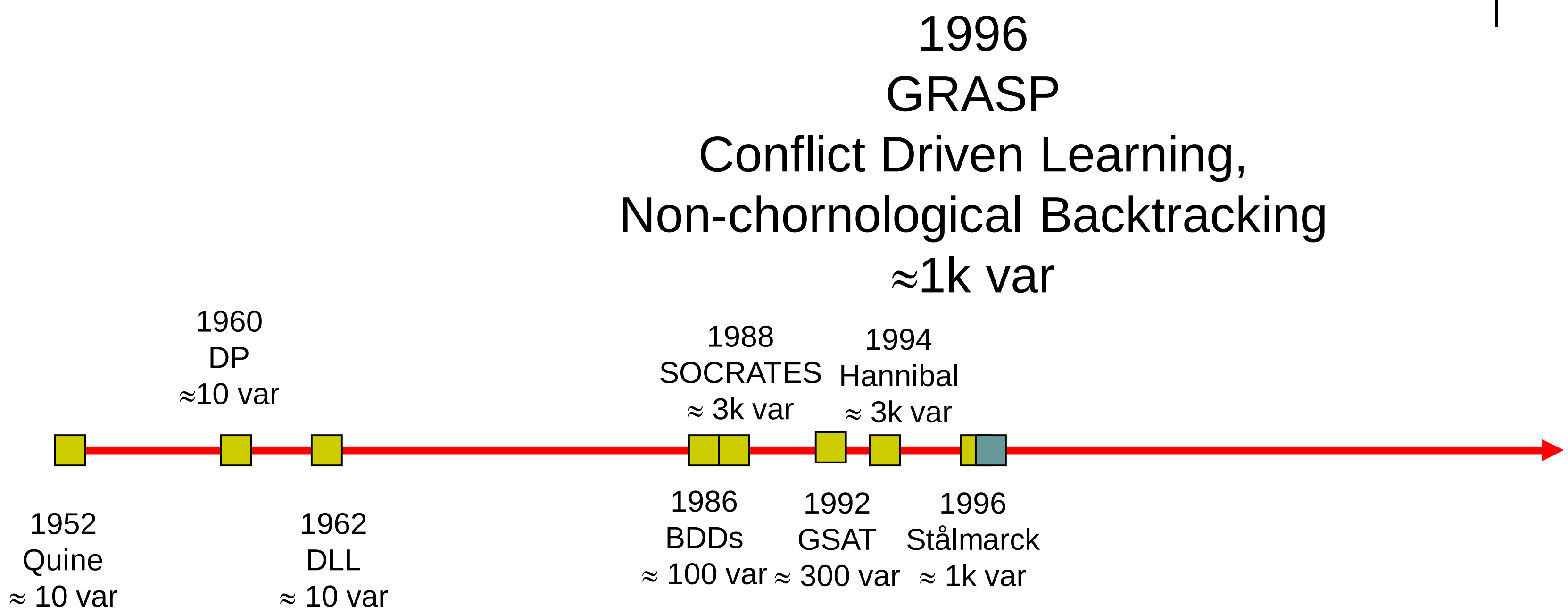
R. Bryant. “Graph-based algorithms for Boolean function manipulation”. *IEEE Trans. on Computers*, C-35, 8:677-691, 1986.

- Store the function in a Directed Acyclic Graph (DAG) representation.
  - Compacted form of the function decision tree.
- Reduction rules guarantee canonicity under fixed variable order.
- Provides for efficient Boolean function manipulation.
- Overkill for SAT.

The Timeline



The Timeline



GRASP

- Marques-Silva and Sakallah [SS96,SS99]

  J. P. Marques-Silva and K. A. Sakallah, "GRASP -- A New Search Algorithm for Satisfiability,“ Proc. ICCAD 1996.

  J. P. Marques-Silva and Karem A. Sakallah, “GRASP: A Search Algorithm for Propositional Satisfiability”, *IEEE Trans. Computers*, C-48, 5:506-521, 1999.
- Incorporates conflict driven learning and non-chronological backtracking
- Practical SAT instances can be solved in reasonable time
- Bayardo and Schrag’s RelSAT also proposed conflict driven learning [BS97]

  R. J. Bayardo Jr. and R. C. Schrag “Using CSP look-back techniques to solve real world SAT instances.” *Proc. AAAI*, pp. 203-208, 1997(144 citations)

Implication Graphs

The implication graph tracks how assignments are implied.

Definition (Implication graph)

An *implication graph* is a labeled directed acyclic graph $G = (V, E)$ where

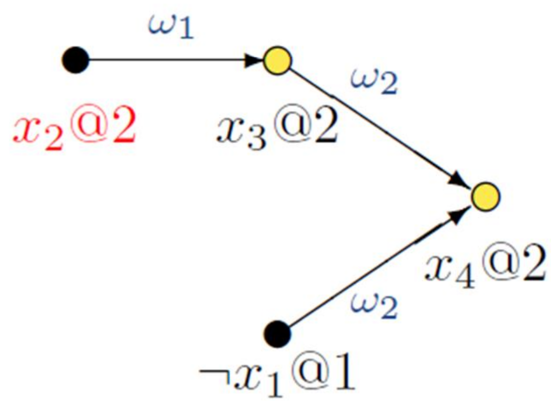
- V: literals of the current partial assignment.
  Labeled with the literal and the decision level.
- E: labeled with the clause that caused the implication.
- Can also contain a single conflict node labeled with κ and incoming edges labeled with some conflicting clause.

A Small Implication Graph Example

Current truth assignment: $\{\neg x_1 @ 1\}$

Decision: $x_2 @ 2$

Clauses
$\omega_1 = (\neg x_2 \vee x_3)$
$\omega_2 = (x_1 \vee \neg x_3 \vee x_4)$

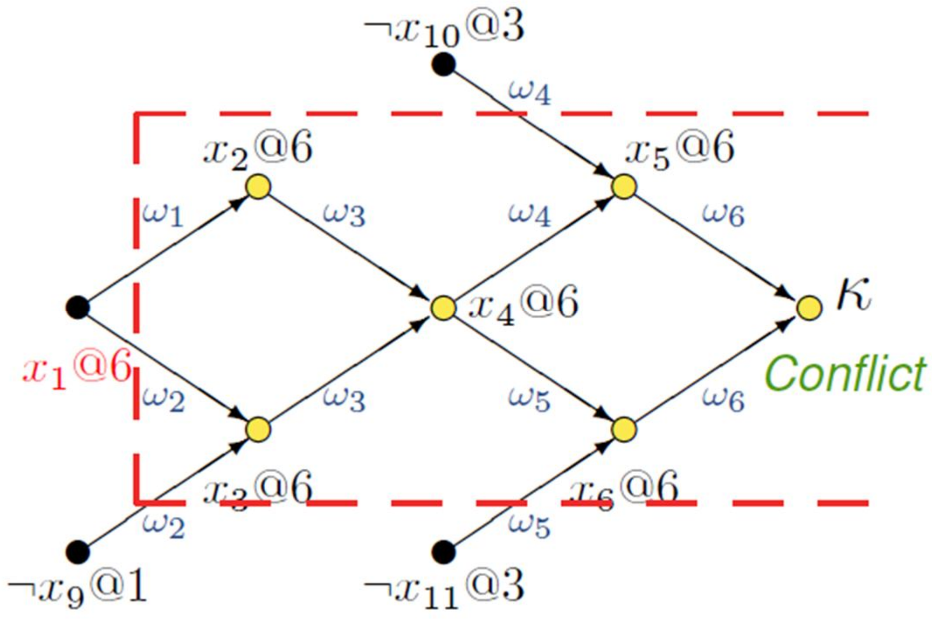


Implication Graphs and Learning

Current truth assignment: $\{\neg x_9@1,\ \neg x_{10}@3,\ \neg x_{11}@3,\ x_{12}@2,\ x_{13}@2\}$

Decision: $x_1@6$

Clauses		
ω_1	$=$	$(\neg x_1 \vee x_2)$
ω_2	$=$	$(\neg x_1 \vee x_3 \vee x_9)$
ω_3	$=$	$(\neg x_2 \vee \neg x_3 \vee x_4)$
ω_4	$=$	$(\neg x_4 \vee x_5 \vee x_{10})$
ω_5	$=$	$(\neg x_4 \vee x_6 \vee x_{11})$
ω_6	$=$	$(\neg x_5 \vee \neg x_6)$
ω_7	$=$	$(x_1 \vee x_7 \vee \neg x_{12})$
ω_8	$=$	$(x_1 \vee x_8)$
ω_9	$=$	$(\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
ω_{10}	$=$	$(\neg x_1 \vee x_9 \vee x_{11} \vee x_{10})$



We learn the *conflict clause*

$\omega_{10} = (\neg x_1 \vee x_9 \vee x_{11} \vee x_{10})$

What can we learn from the implication graph?

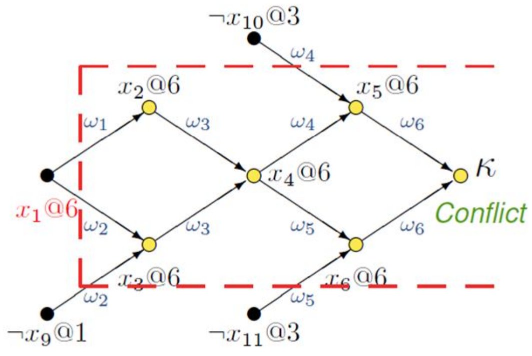
- The decisions causing conflicts are the roots. i.e., the sinks
- None of the other decisions, e.g., at levels 4,5, matter
- Learn you can't have assignment: $(x_1 . x_9' . x_{10}' . x_{11}')$

Implication Graphs and Learning

Current truth assignment: $\{\neg x_9@1,\ \neg x_{10}@3,\ \neg x_{11}@3,\ x_{12}@2,\ x_{13}@2\}$

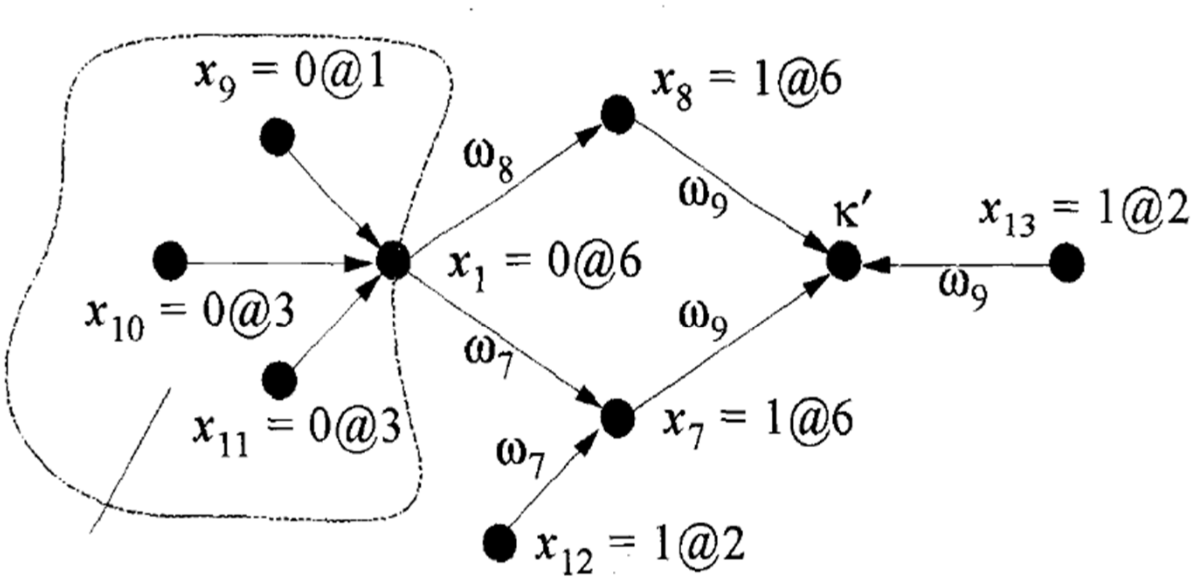
Decision: $x_1@6$

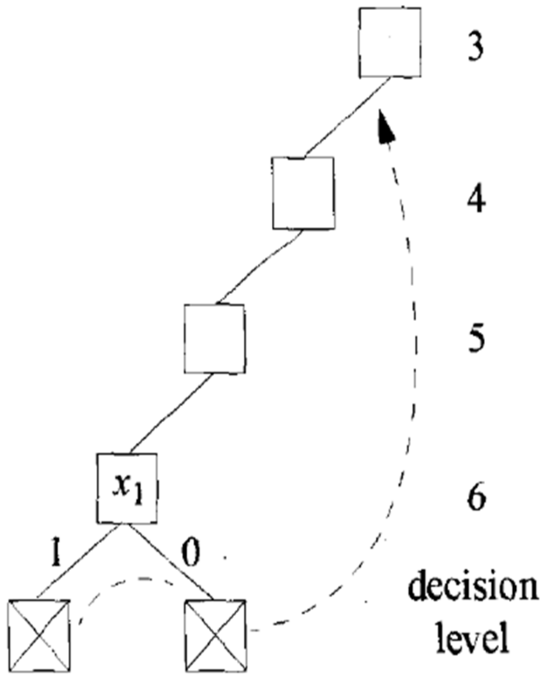
Clauses		
ω_1	=	$(\neg x_1 \vee x_2)$
ω_2	=	$(\neg x_1 \vee x_3 \vee x_9)$
ω_3	=	$(\neg x_2 \vee \neg x_3 \vee x_4)$
ω_4	=	$(\neg x_4 \vee x_5 \vee x_{10})$
ω_5	=	$(\neg x_4 \vee x_6 \vee x_{11})$
ω_6	=	$(\neg x_5 \vee \neg x_6)$
ω_7	=	$(x_1 \vee x_7 \vee \neg x_{12})$
ω_8	=	$(x_1 \vee x_8)$
ω_9	=	$(\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
ω_{10}	=	$(\neg x_1 \vee x_9 \vee x_{11} \vee x_{10})$



We learn the *conflict clause*
$\omega_{10} = (\neg x_1 \vee x_9 \vee x_{11} \vee x_{10})$

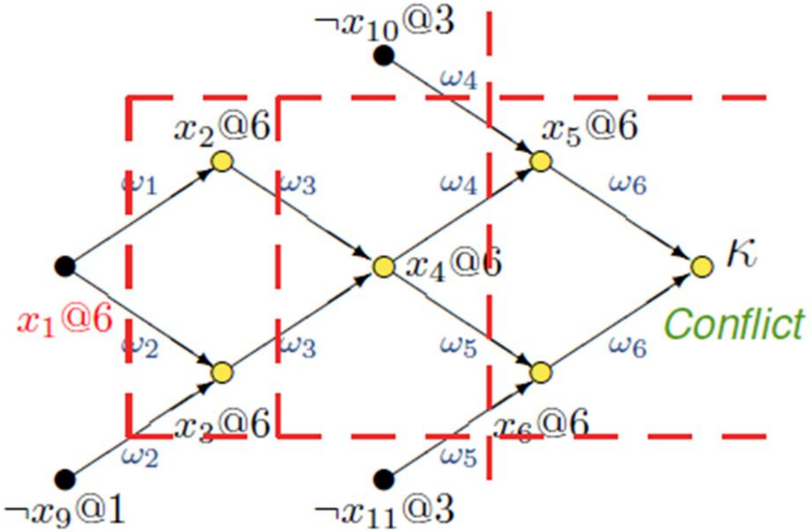
backtrack to the largest decision level in the conflict Clause !!





More Conflict Clauses

- Def.: A *conflict clause* is any clause implied by the formula



$$\neg x_1 \vee x_9 \vee x_{10} \vee x_{11}$$

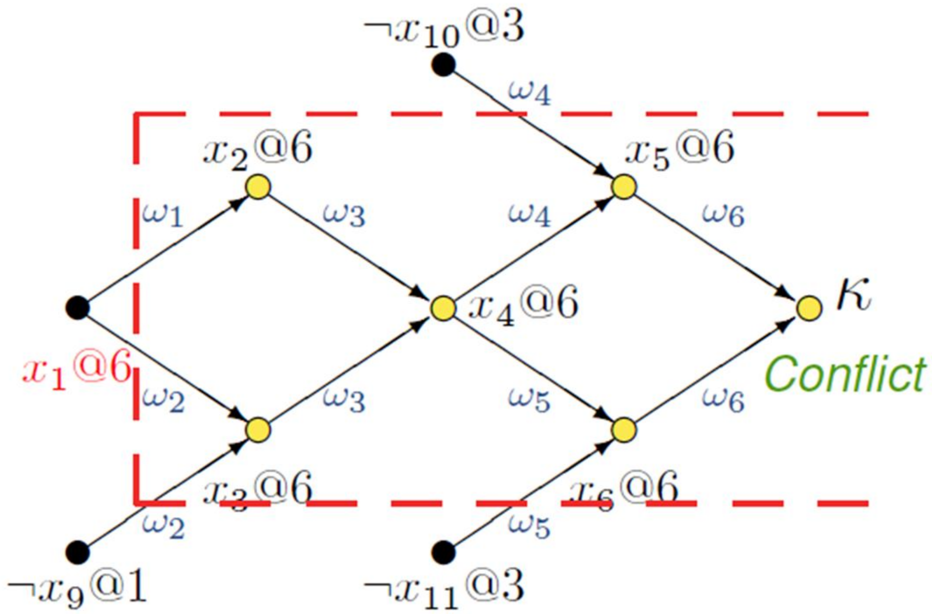
$$\neg x_2 \vee \neg x_3 \vee x_{10} \vee x_{11}$$

$$\neg x_4 \vee x_{10} \vee x_{11}$$

What constitutes a sufficient condition for conflict:

1. Make a cut to separate "conflict side" from "reason side"
2. Conjunction (C) of literals labeling the set of nodes on the reason side that have at least one edge to the conflict side

Conflict Clause: Negation of C

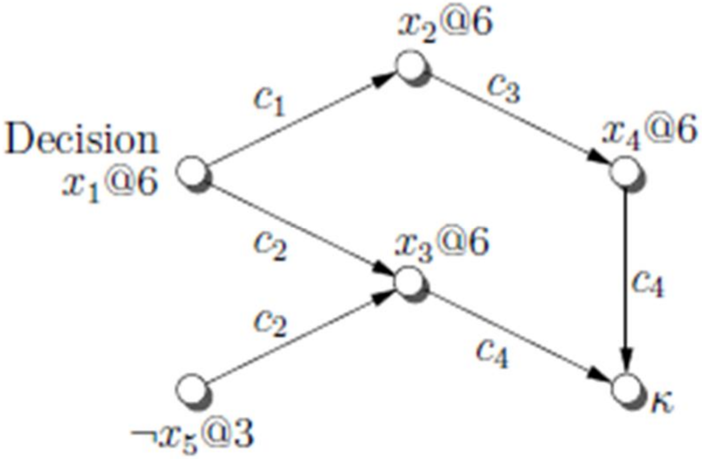


What is a good conflict clause?

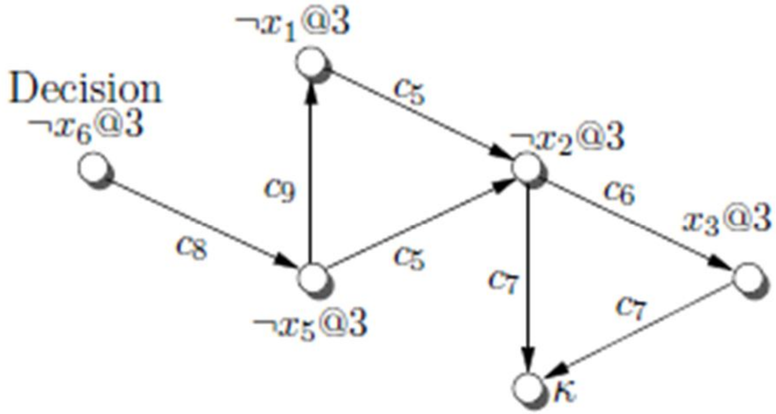
- Must be new!!
- An "asserting clause" that flips decision made at current level
- Backtracks to the lowest level?
- Shorter clauses

Definition 2.9 (unique implication point (UIP)). *Given a partial conflict graph corresponding to the decision level of the conflict, a* unique implication point (UIP) *is any node other than the conflict node that is on all paths from the decision node to the conflict node.*

Definition 2.10 (first UIP). *A* first UIP *is a UIP that is closest to the conflict node.*



Decision
$x_1@6$
$x_2@6$
$x_3@6$
$x_4@6$
$\neg x_5@3$
c_1
c_2
c_2
c_3
c_4
c_4
κ



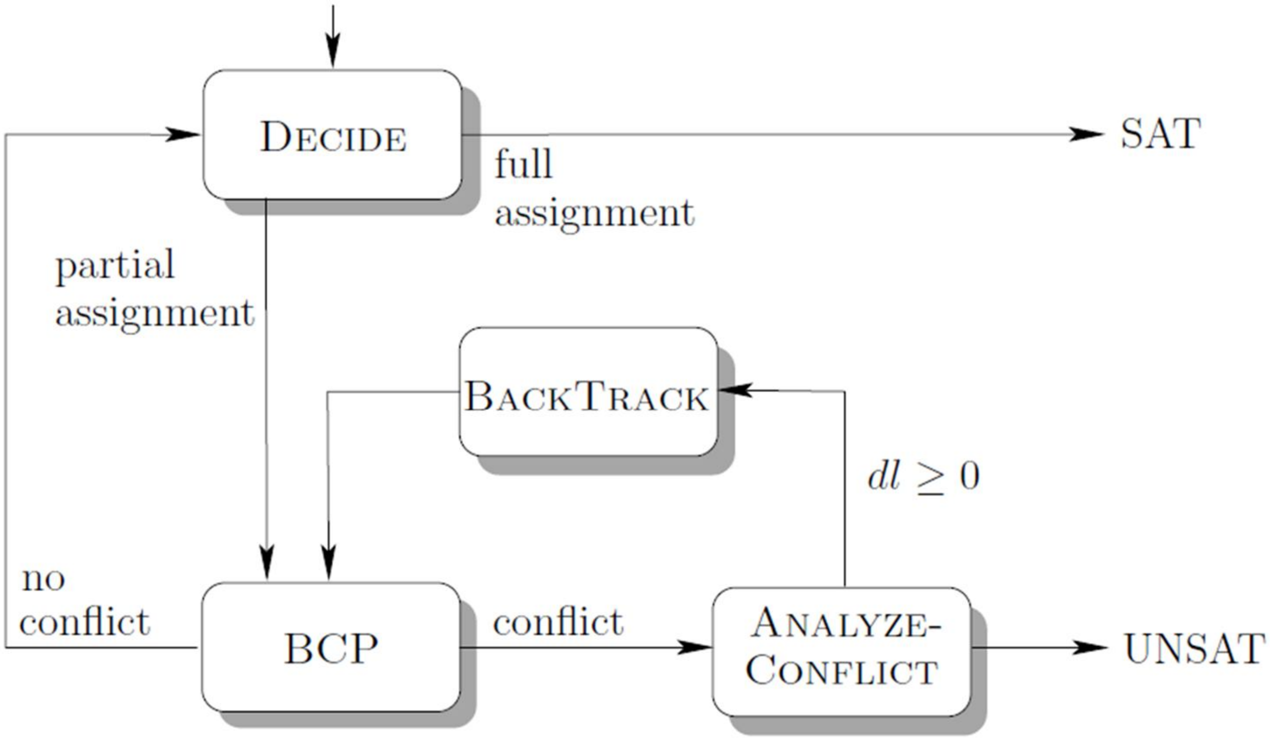
Decision
$\neg x_6@3$
$\neg x_1@3$
$\neg x_2@3$
$x_3@3$
$\neg x_5@3$
c_5
c_5
c_6
c_7
c_7
c_8
c_9
κ

Basic DPLL

Given the partial assignment

$$\{x_1 \mapsto 1,\ x_2 \mapsto 0,\ x_4 \mapsto 1\}$$

$(x_1 \vee x_3 \vee \neg x_4)$	is satisfied
$(\neg x_1 \vee x_2)$	is conflicting
$(\neg x_1 \vee \neg x_4 \vee x_3)$	is unit
$(\neg x_1 \vee x_3 \vee x_5)$	is unresolved.



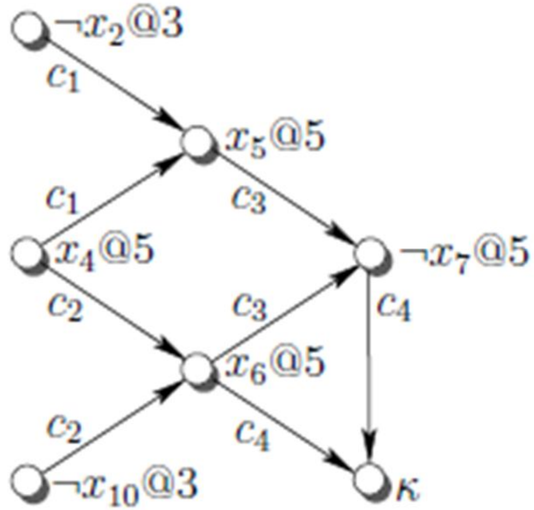
Algorithm 2.2.2: ANALYZE-CONFLICT

Input:
Output: Backtracking decision level + a new conflict clause

```
if current-decision-level = 0 then return -1;
cl := current-conflicting-clause;
while (¬STOP-CRITERION-MET(cl)) do
      lit := LAST-ASSIGNED-LITERAL(cl);
      var := VARIABLE-OF-LITERAL(lit);
      ante := ANTECEDENT(lit);
      cl := RESOLVE(cl, ante, var);
add-clause-to-database(cl);
return clause-asserting-level(cl);          ▷ 2nd highest decision level in cl
```

Definition 2.9 (unique implication point (UIP)). *Given a partial conflict graph corresponding to the decision level of the conflict, a* unique implication point (UIP) *is any node other than the conflict node that is on all paths from the decision node to the conflict node.*

$$
\begin{aligned}
c_1 &= (\neg x_4 \vee x_2 \vee x_5) \\
c_2 &= (\neg x_4 \vee x_{10} \vee x_6) \\
c_3 &= (\neg x_5 \vee \neg x_6 \vee \neg x_7) \\
c_4 &= (\neg x_6 \vee x_7) \\
&\vdots
\end{aligned}
$$



Decision Heuristics: DLIS

DLIS (Dynamic Largest Individual Sum)

choose the assignment that increases the number of satisfied clauses the most

- C_{xp}: number of unresolved clauses in which x appears positively
- C_{xn}: number of unresolved clauses in which x appears negatively
- Let x be the literal for which C_{xp} is maximal
- Let y be the literal for which C_{yn} is maximal

- If $C_{xp} > C_{yn}$, choose x and assign TRUE
- Otherwise, choose y and assign FALSE

Decision Heuristics: JW

Jeroslow-Wang method

For every clause ω and every literal l, compute:

$$J(l) = \sum_{l \in \omega, \omega \in \varphi} 2^{-|\omega|}$$

- $|\omega|$ is the length of the clause (count the literals)
- Make decision l that maximizes $J(l)$

- This gives exponentially higher weight to literals in shorter clauses
- Can be done dynamically (only for unresolved clauses) or upfront

Decision Heuristics: VSIDS

VSIDS (Variable State Independent Decaying Sum)

1. Each variable in each polarity has a counter initialized to 0.
2. When a clause is added, the counters are updated.
3. The unassigned variable with the highest counter is chosen.
4. Periodically, all the counters are divided by a constant.

$\Rightarrow$ variables appearing in recent conflicts get higher priority

Decision Heuristics: VSIDS

- Keep a list of variables/polarities
- Updates only needed when adding a conflict clause
- Decisions are made in constant time (how?)

Decision Heuristics: VSIDS

VSIDS is a ‘quasi-static’ strategy:

- static as it does not depend on the current assignment
- dynamic as the weights change over time

VSIDS is called a *conflict-driven* decision strategy.

”...this strategy dramatically (i.e., an order of magnitude) improved performance...”