

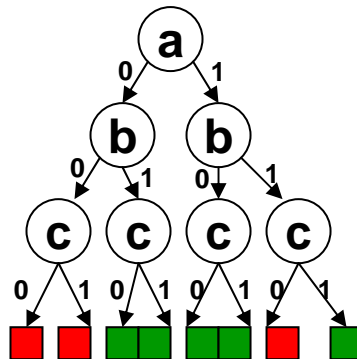
Satisfiability of Propositional Formulas

Mooly Sagiv

Based on a presentation by Sharad Malik & Ohad Shacham

The SAT Problem

- Given a propositional formula (Boolean function)
 - $\varphi = (\mathbf{a} \vee \mathbf{b}) \wedge (\neg \mathbf{a} \vee \neg \mathbf{b} \vee \mathbf{c})$
- Determine if φ is valid
- Determine if φ is satisfiable
 - Find a satisfying assignment or report that such does not exist
- For n variables, there are 2^n possible truth assignments to be checked



Why Bother?

- Core computational engine for major applications
 - Artificial Intelligence
 - Knowledge base deduction
 - Automatic theorem proving
 - Electronic Design Automaton
 - Testing and Verification
 - Logic synthesis
 - FPGA routing
 - Path delay analysis
 - And more...

Problem Representation

- Represent the formulas in Conjunctive Normal Form (CNF)
- Conversion to CNF is straightforward
 - $\mathbf{a \vee (b \wedge \neg(c \vee \neg d)) \equiv (a \vee (b \wedge \neg c \wedge \neg \neg d)) \equiv (a \vee (b \wedge \neg c \wedge d)) \equiv (a \vee b) \wedge (a \vee \neg c) \wedge (a \vee d)}$
 - **May need to add variables**
- Notations
 - Literals
 - Variable or its negation
 - Clauses
 - Disjunction of literals
 - $\mathbf{\varphi = (a \vee b) \wedge (\neg a \vee \neg b \vee c) \equiv (a + b)(a' + b' + c)}$
- Advantages of CNF
 - Simple data structure
 - All the clauses need to be satisfied

Complexity Results

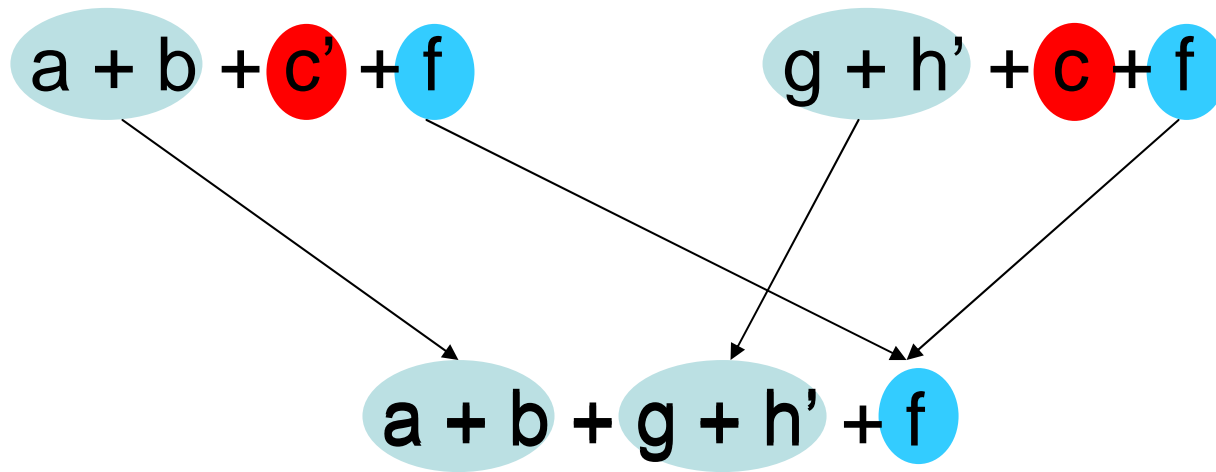
- First established NP-Complete problem
 - Even when at most 3 literals per clause (3-SAT)
 - S. A. Cook, The complexity of theorem proving procedures, *Proceedings, Third Annual ACM Symp. on the Theory of Computing*, 1971, 151-158
 - No polynomial algorithm for all instances unless $P = NP$
- Becomes polynomial when
 - At most two literals per clause (2-SAT)
 - At most one positive literal in every clause (Horn)

Goals

- Develop algorithms which solve all SAT instances
- Exponential worst case complexity
- But works well on many instances
 - Interesting Heuristics
 - Annual SAT conferences
 - SAT competitions
 - Randomly, Handmade, Industrial, AI
 - 10 Millions variables!

Resolution

- Resolution of a pair of clauses with exactly ONE incompatible variable

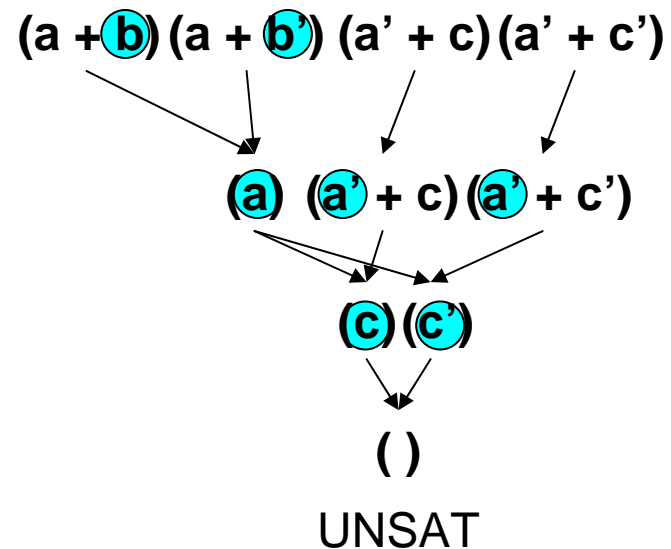
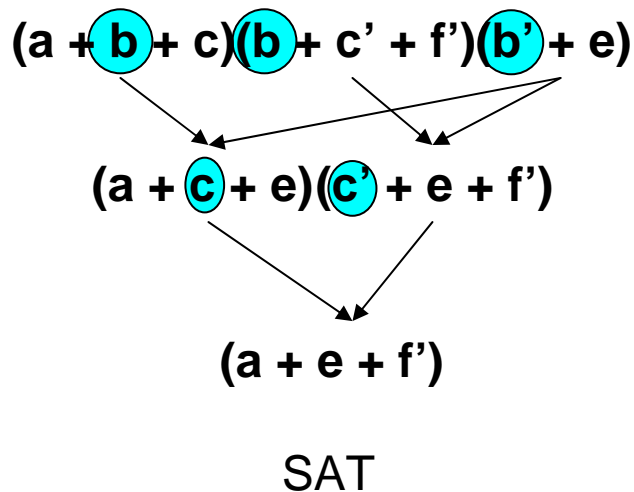


- What if more than one incompatible variables?

Davis Putnam Algorithm

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

- Iteratively select a variable for resolution till no more variables are left
- Report UNSAT when the empty clause occurs
- Can discard resolved clauses after each iteration



Potential memory explosion problem!

Can we avoid using exponential
space?

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

- Basic framework for many modern SAT solvers
- Also known as DPLL for historical reasons

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)

a

Basic DLL Procedure - DFS

(a' + b + c)

(a + c + d)

(a + c + d')

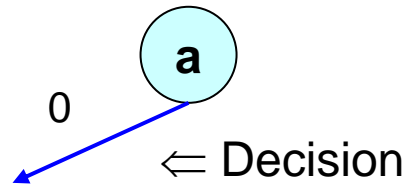
(a + c' + d)

(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')

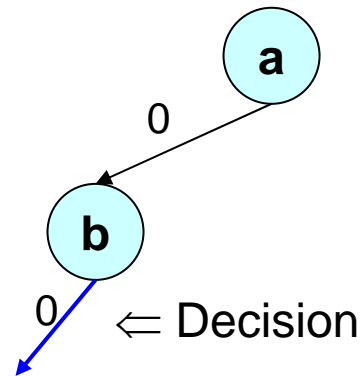
(a + c' + d)

(a + c' + d')

(b' + c' + d)

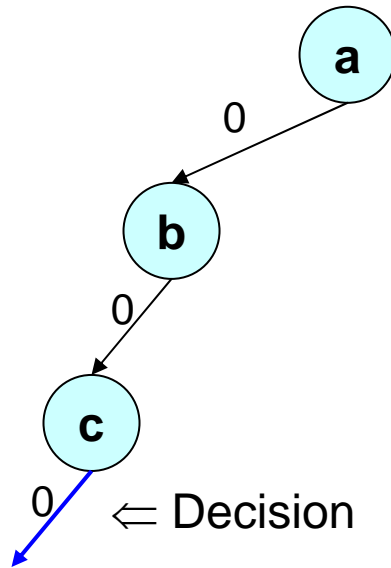
(a' + b + c')

(a' + b' + c)



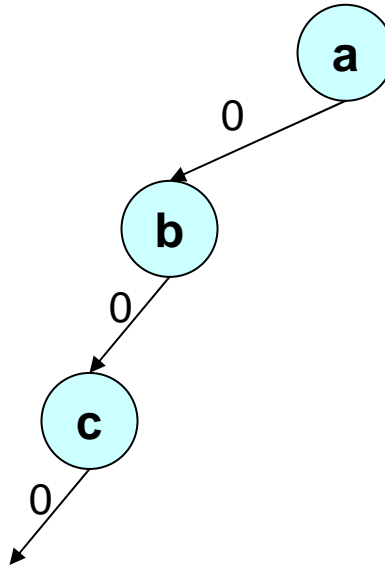
Basic DLL Procedure - DFS

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

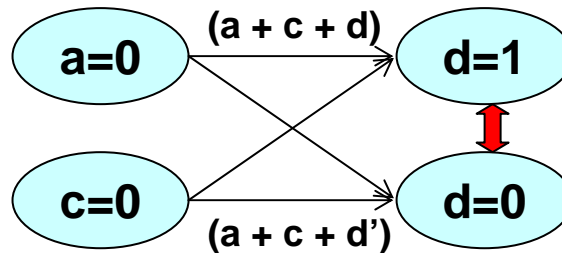


Basic DLL Procedure - DFS

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



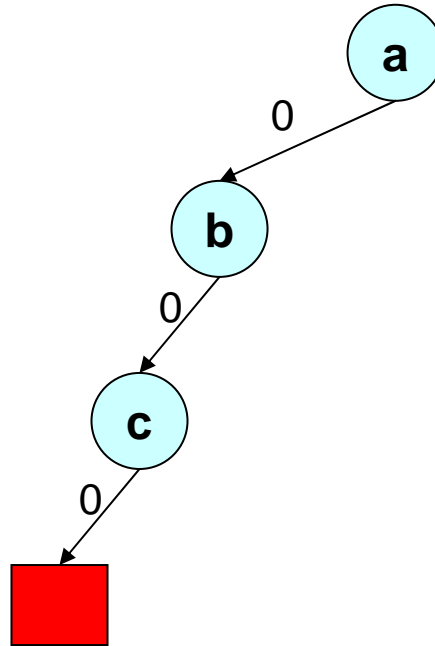
Implication Graph



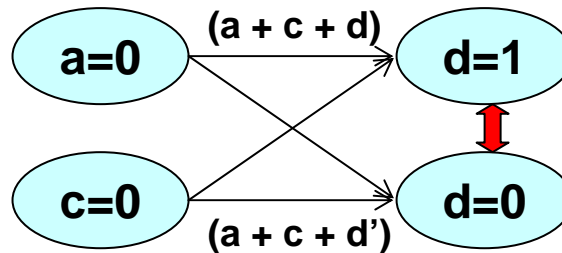
Conflict!

Basic DLL Procedure - DFS

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



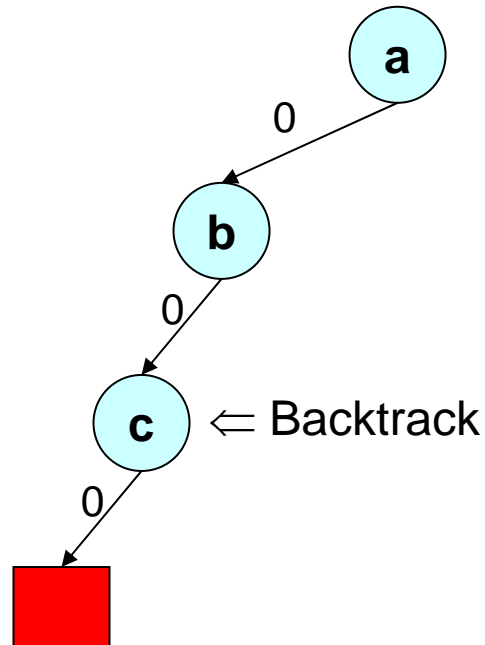
Implication Graph



Conflict!

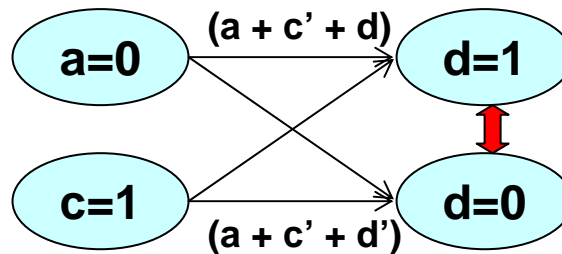
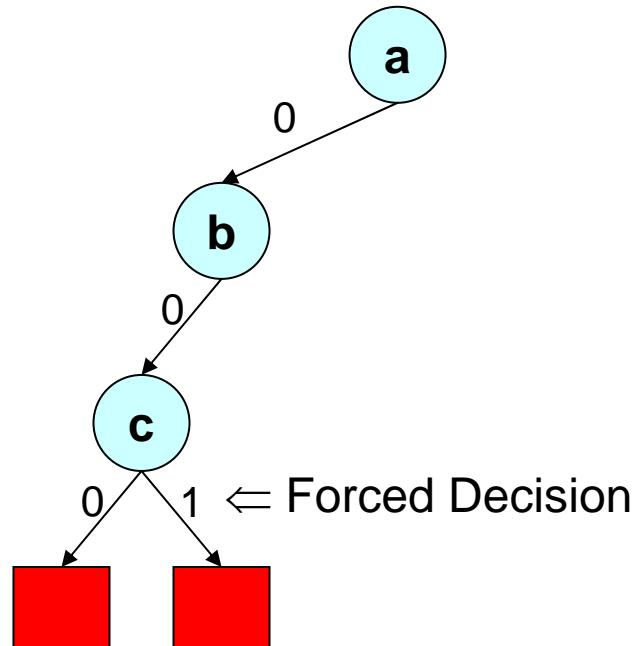
Basic DLL Procedure - DFS

(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

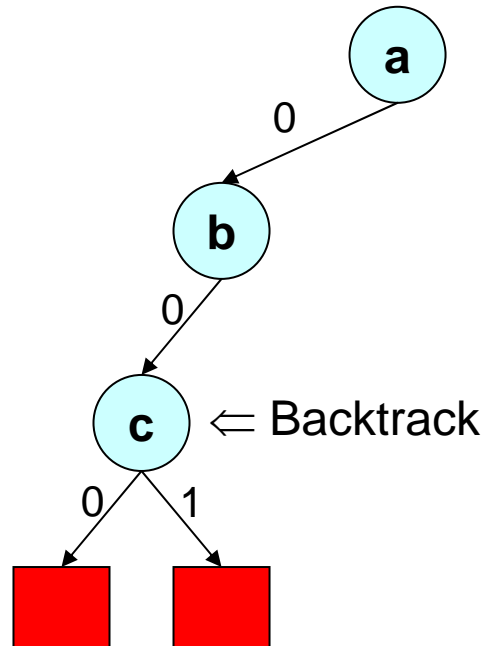
$(a' + b + c)$
 $(a + c + d)$
 $(a + c + d')$
 $(a + c' + d)$
 $(a + c' + d')$
 $(b' + c' + d)$
 $(a' + b + c')$
 $(a' + b' + c)$



Conflict!

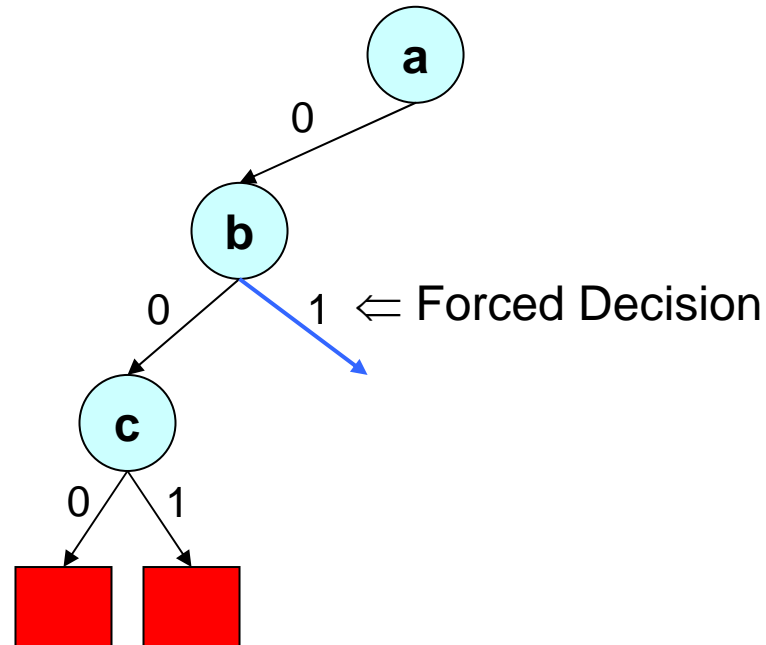
Basic DLL Procedure - DFS

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



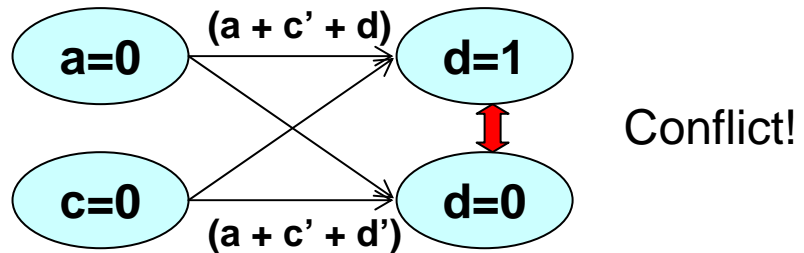
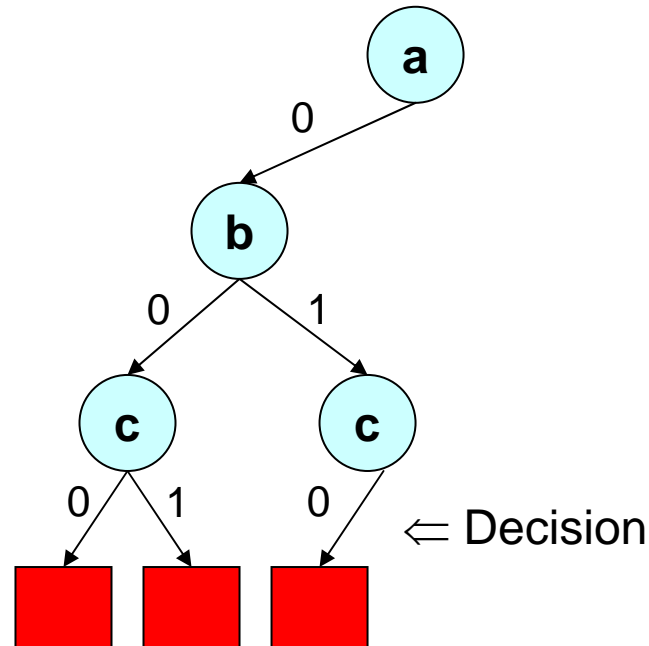
Basic DLL Procedure - DFS

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



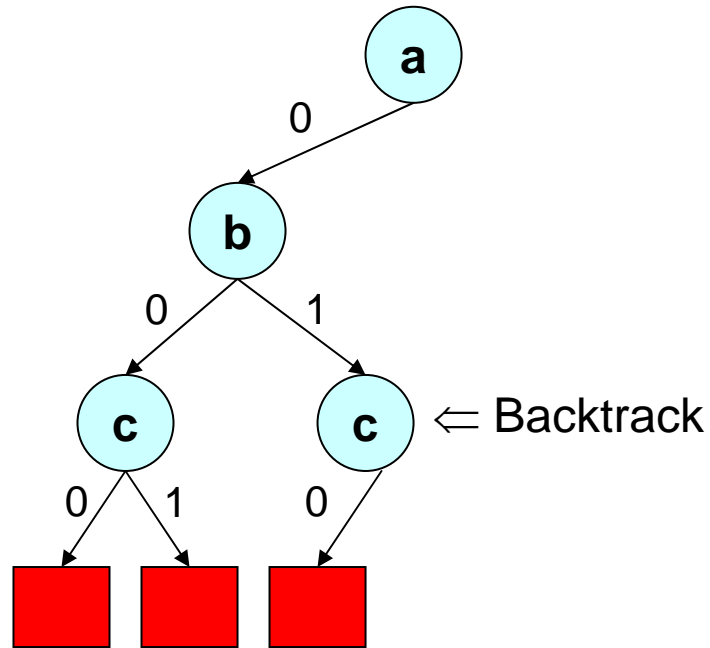
Basic DLL Procedure - DFS

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



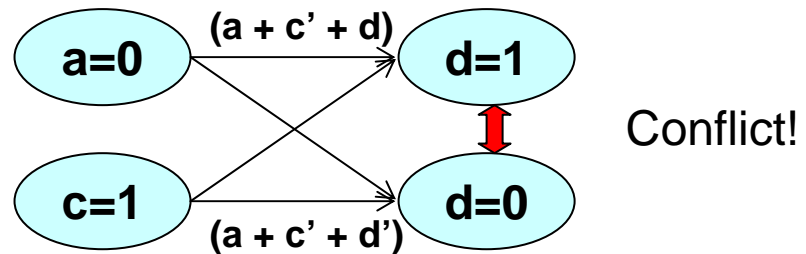
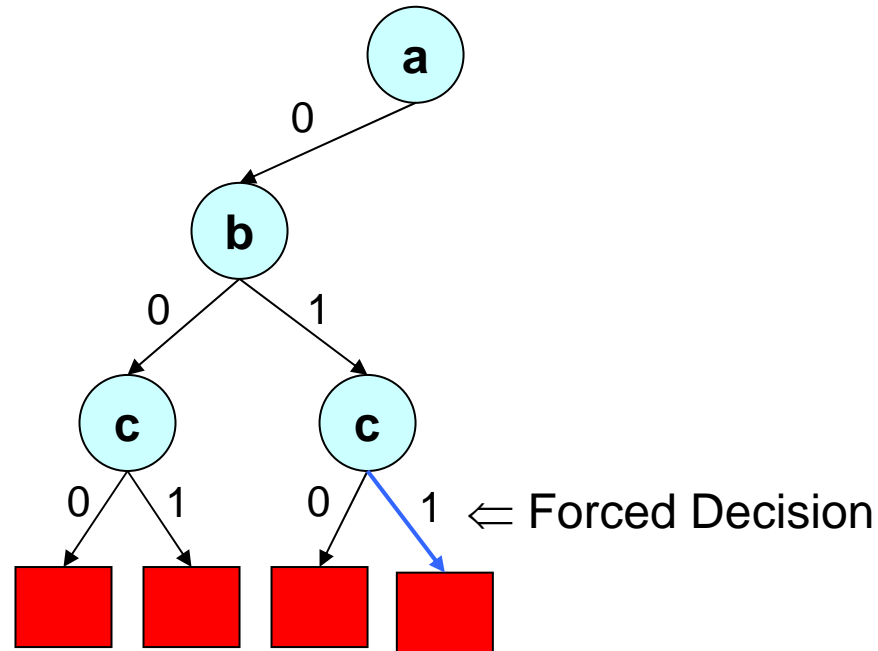
Basic DLL Procedure - DFS

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



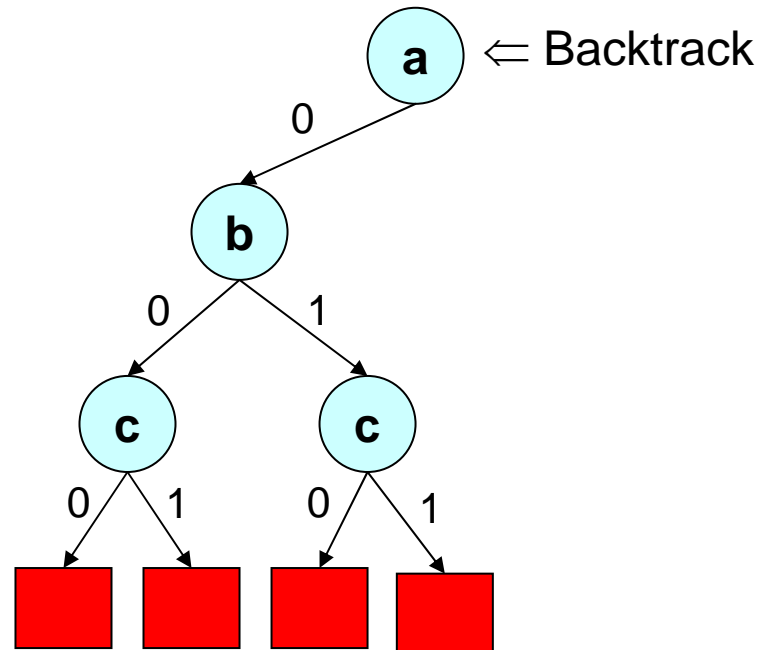
Basic DLL Procedure - DFS

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



Basic DLL Procedure - DFS

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



Basic DLL Procedure - DFS

$(a' + b + c)$

$(a + c + d)$

$(a + c + d')$

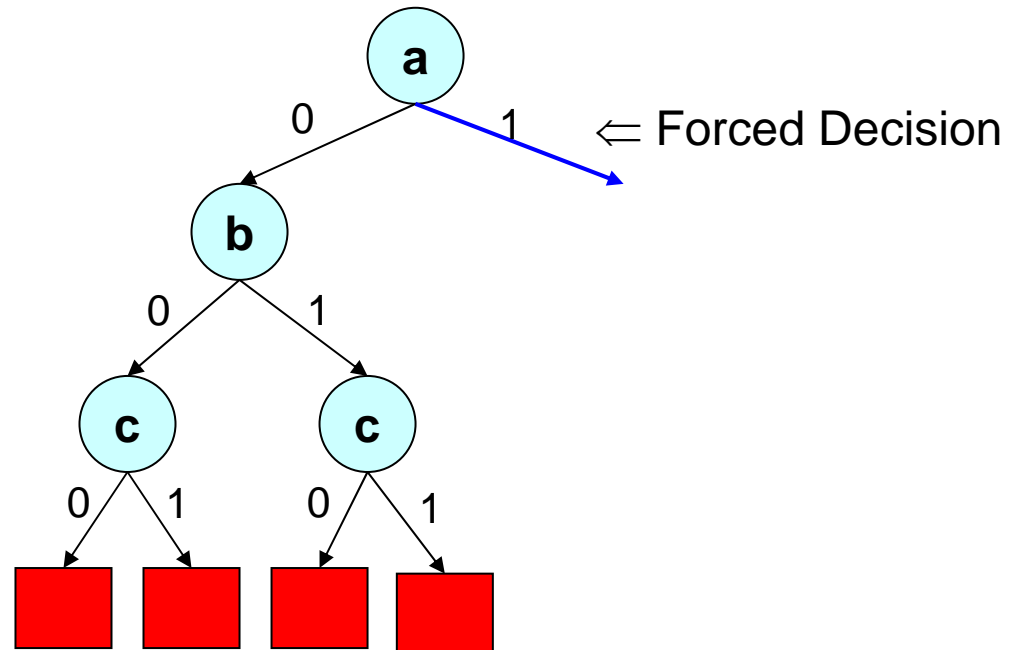
$(a + c' + d)$

$(a + c' + d')$

$(b' + c' + d)$

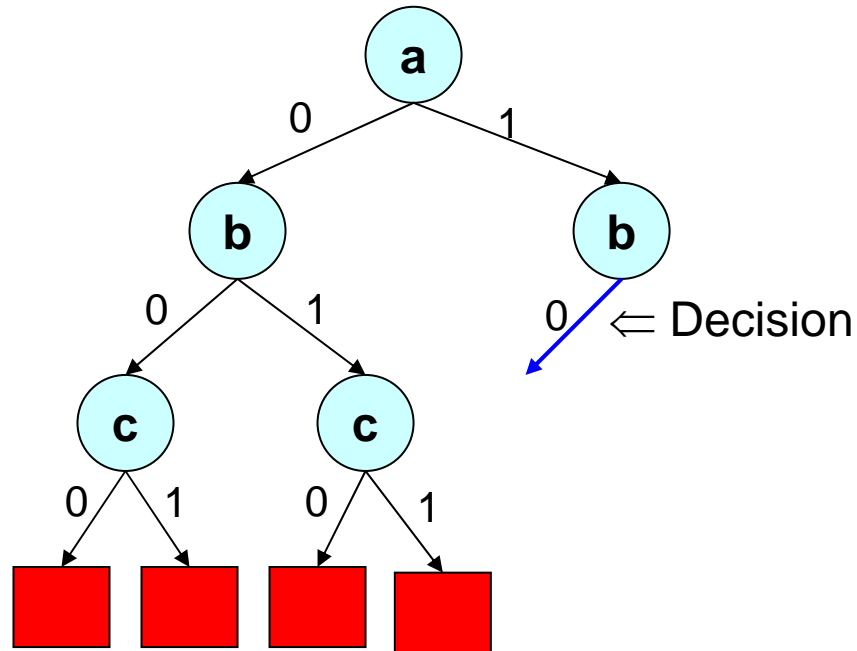
$(a' + b + c')$

$(a' + b' + c)$



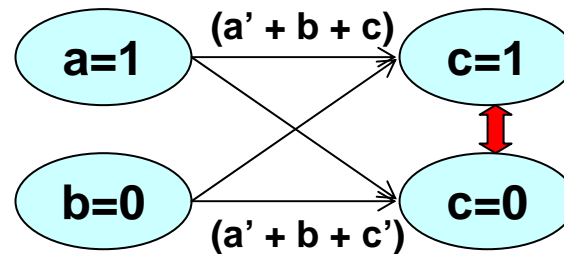
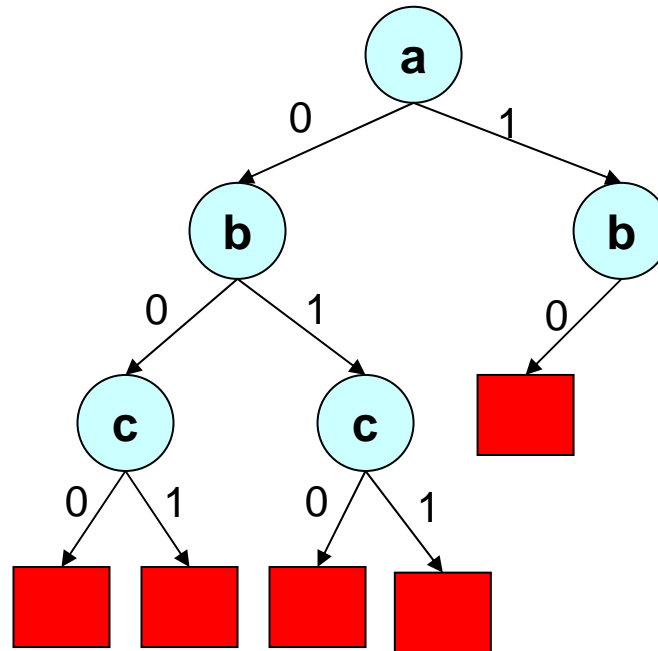
Basic DLL Procedure - DFS

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



Basic DLL Procedure - DFS

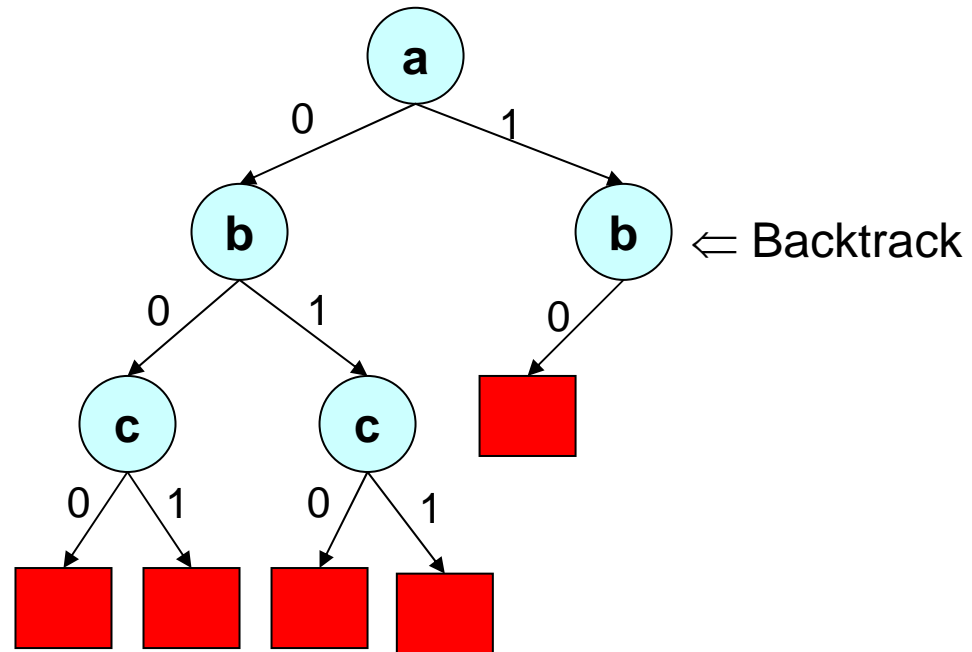
$(a' + b + c)$
 $(a + c + d)$
 $(a + c + d')$
 $(a + c' + d)$
 $(a + c' + d')$
 $(b' + c' + d)$
 $(a' + b + c')$
 $(a' + b' + c)$



Conflict!

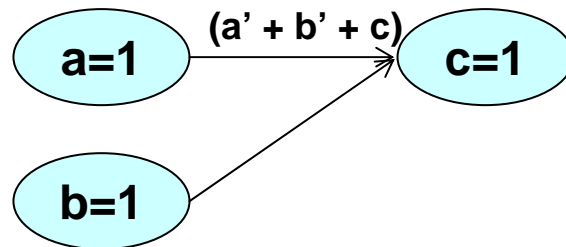
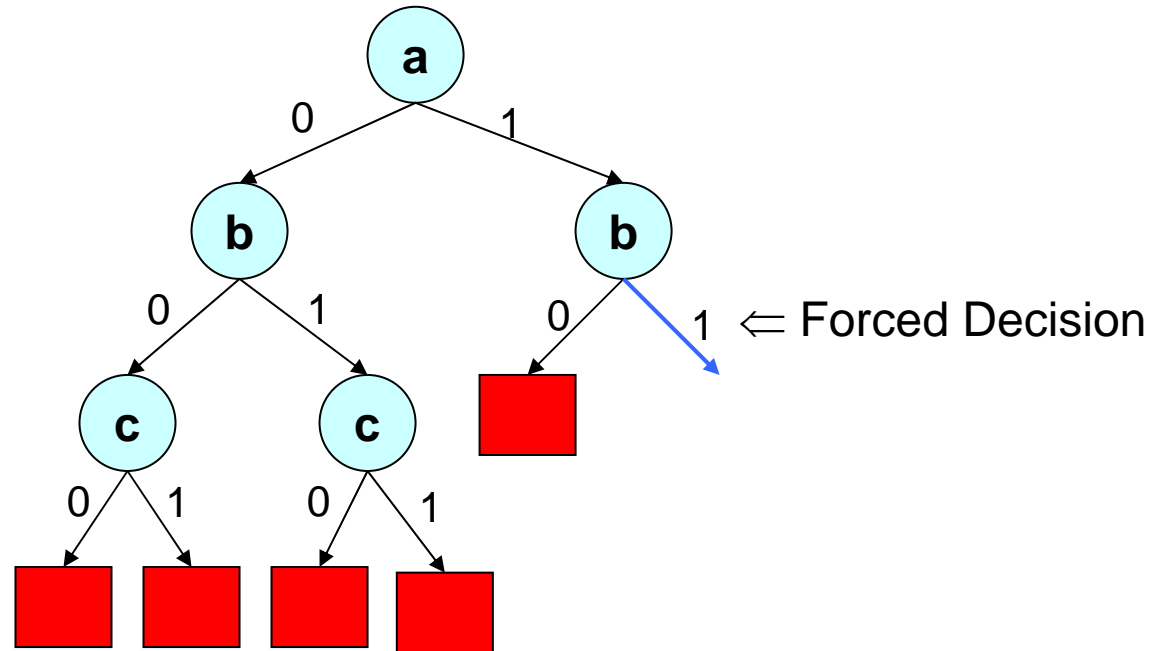
Basic DLL Procedure - DFS

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



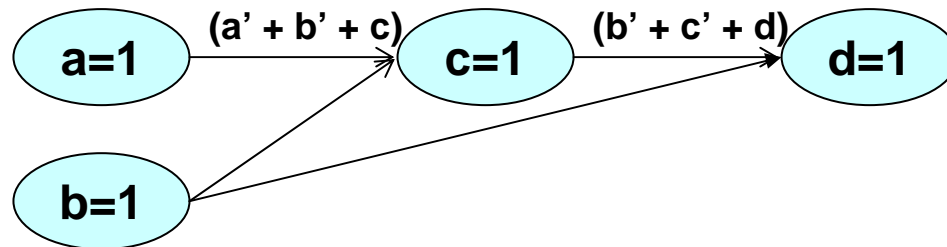
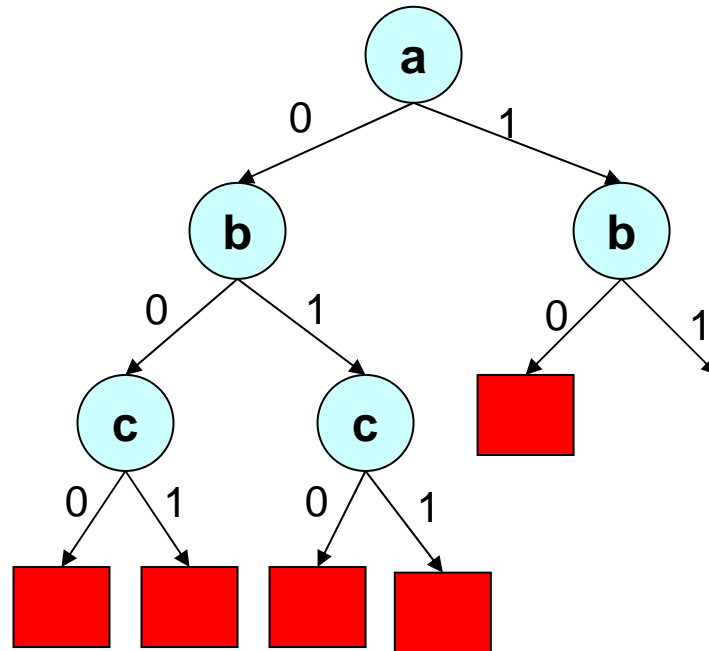
Basic DLL Procedure - DFS

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



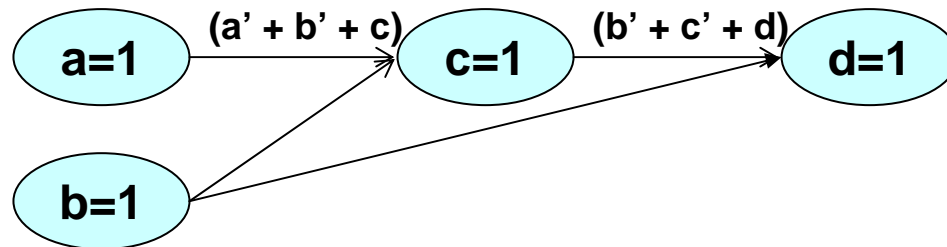
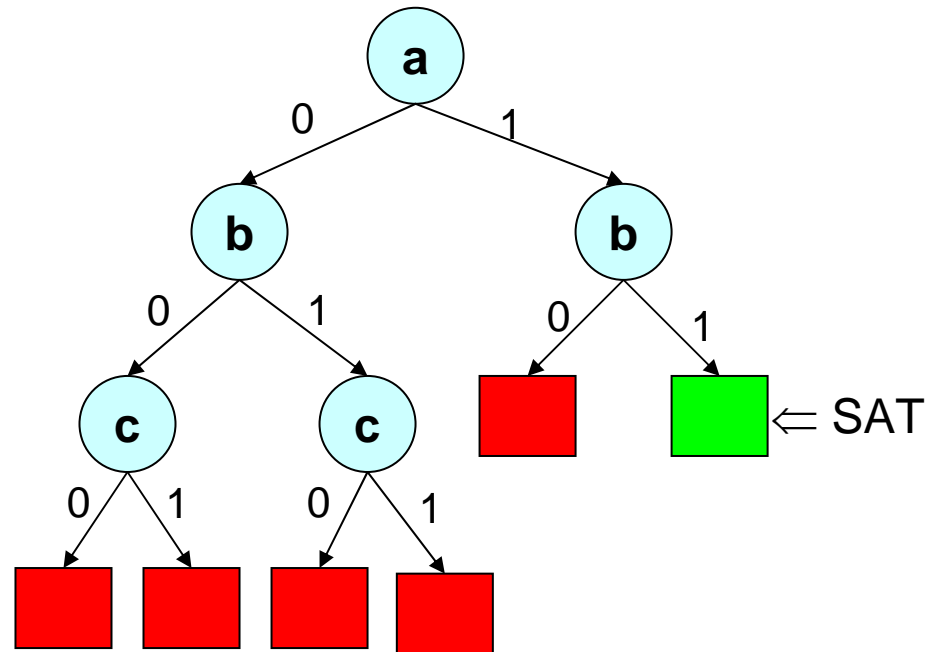
Basic DLL Procedure - DFS

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



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 DLL

- 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)

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

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

$$a = T, b = T, c \text{ is unassigned}$$

Satisfied Literal

Unsatisfied Literal

Unassigned Literal

- 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.
- Workhorse of DLL based algorithms

GRASP

- Marques-Silva and Sakallah [SS96,SS99]
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 ReISAT 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

Conflict Driven Learning and Non-chronological Backtracking

$x_1 + x_4$

$x_1 + x_3' + x_8'$

$x_1 + x_8 + x_{12}$

$x_2 + x_{11}$

$x_7' + x_3' + x_9$

$x_7' + x_8 + x_9'$

$x_7 + x_8 + x_{10}'$

$x_7 + x_{10} + x_{12}'$

Conflict Driven Learning and Non-chronological Backtracking

$$x1 + x4$$

$$x1 + x3' + x8'$$

$$x1 + x8 + x12$$

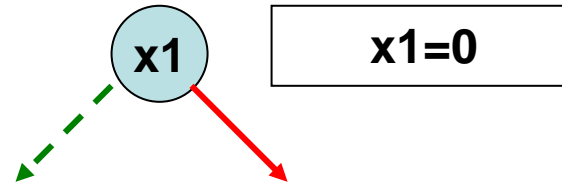
$$x2 + x11$$

$$x7' + x3' + x9$$

$$x7' + x8 + x9'$$

$$x7 + x8 + x10'$$

$$x7 + x10 + x12'$$



○ $x1=0$

Conflict Driven Learning and Non-chronological Backtracking

$$x1 + x4$$

$$x1 + x3' + x8'$$

$$x1 + x8 + x12$$

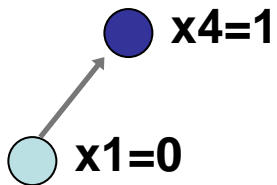
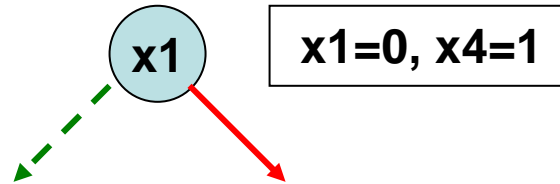
$$x2 + x11$$

$$x7' + x3' + x9$$

$$x7' + x8 + x9'$$

$$x7 + x8 + x10'$$

$$x7 + x10 + x12'$$



Conflict Driven Learning and Non-chronological Backtracking

$x_1 + x_4$

$x_1 + x_3' + x_8'$

$x_1 + x_8 + x_{12}$

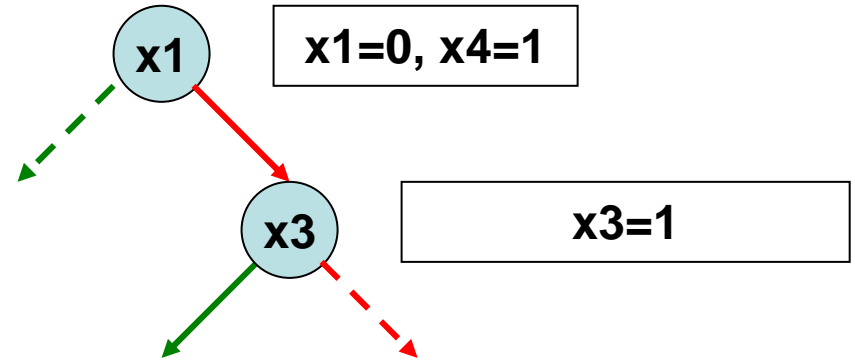
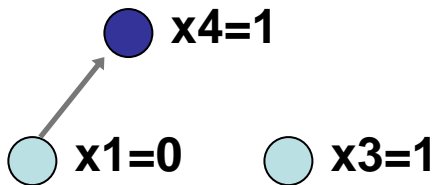
$x_2 + x_{11}$

$x_7' + x_3' + x_9$

$x_7' + x_8 + x_9'$

$x_7 + x_8 + x_{10}'$

$x_7 + x_{10} + x_{12}'$



Conflict Driven Learning and Non-chronological Backtracking

$x_1 + x_4$

$x_1 + x_3' + x_8'$

$x_1 + x_8 + x_{12}$

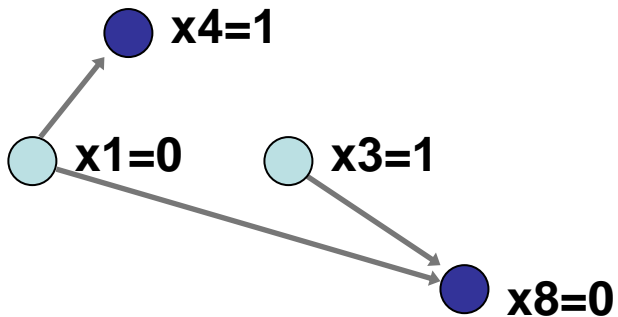
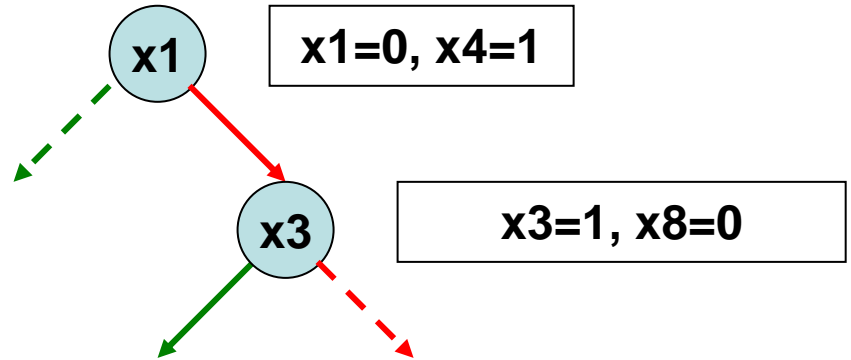
$x_2 + x_{11}$

$x_7' + x_3' + x_9$

$x_7' + x_8 + x_9'$

$x_7 + x_8 + x_{10}'$

$x_7 + x_{10} + x_{12}'$



Conflict Driven Learning and Non-chronological Backtracking

$$x1 + x4$$

$$x1 + x3' + x8'$$

$$x1 + x8 + x12$$

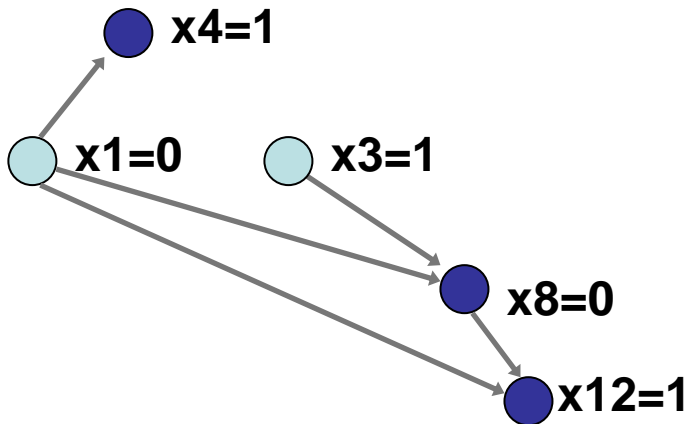
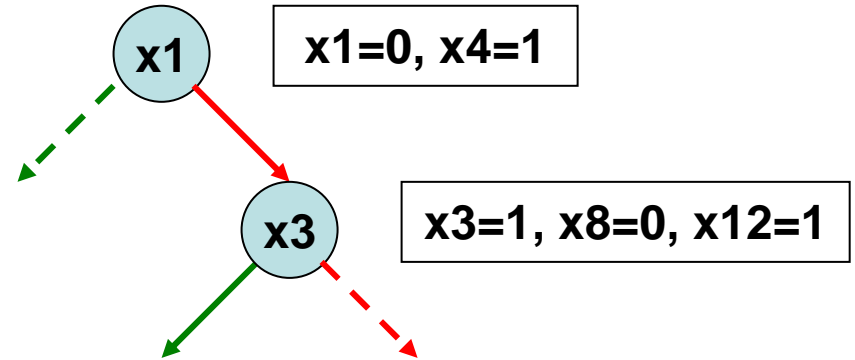
$$x2 + x11$$

$$x7' + x3' + x9$$

$$x7' + x8 + x9'$$

$$x7 + x8 + x10'$$

$$x7 + x10 + x12'$$



Conflict Driven Learning and Non-chronological Backtracking

$$x_1 + x_4$$

$$x_1 + x_3' + x_8'$$

$$x_1 + x_8 + x_{12}$$

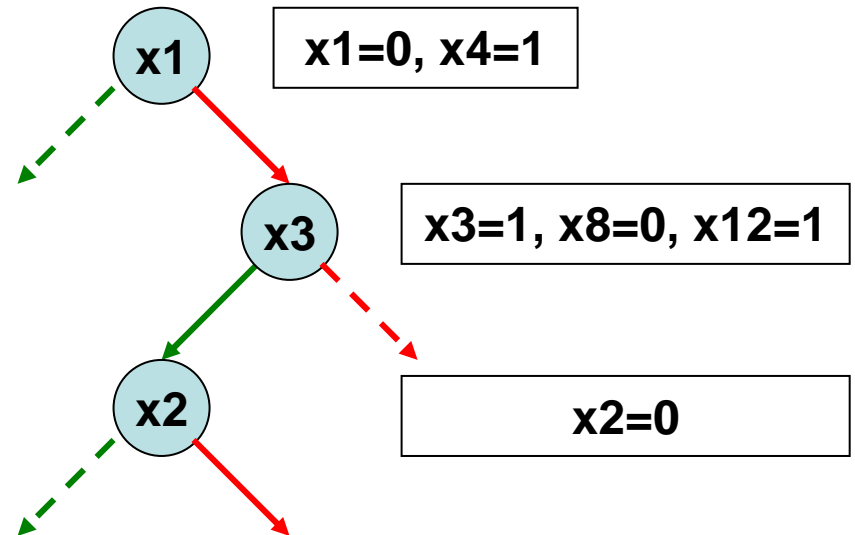
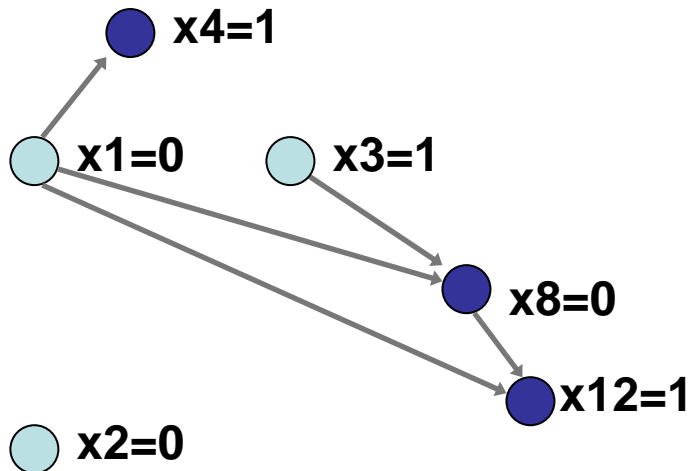
$$x_2 + x_{11}$$

$$x_7' + x_3' + x_9$$

$$x_7' + x_8 + x_9'$$

$$x_7 + x_8 + x_{10}'$$

$$x_7 + x_{10} + x_{12}'$$



Conflict Driven Learning and Non-chronological Backtracking

$$x1 + x4$$

$$x1 + x3' + x8'$$

$$x1 + x8 + x12$$

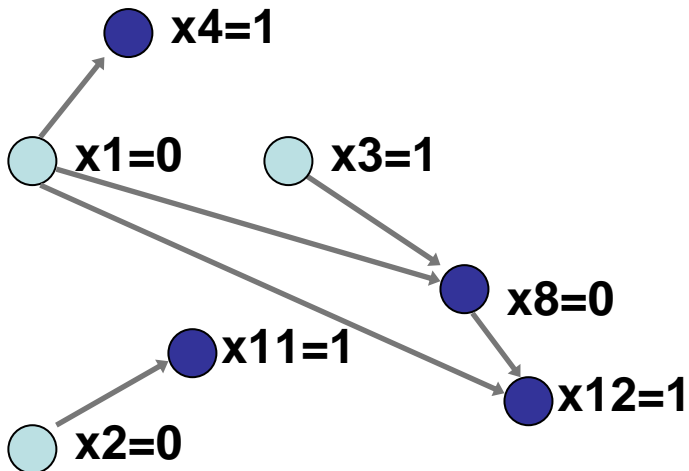
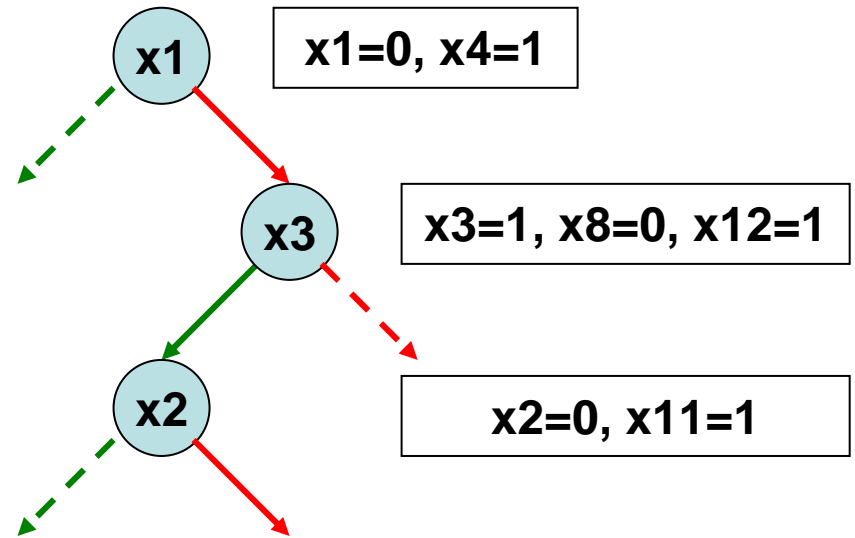
$$x2 + x11$$

$$x7' + x3' + x9$$

$$x7' + x8 + x9'$$

$$x7 + x8 + x10'$$

$$x7 + x10 + x12'$$



Conflict Driven Learning and Non-chronological Backtracking

$x_1 + x_4$

$x_1 + x_3' + x_8'$

$x_1 + x_8 + x_{12}$

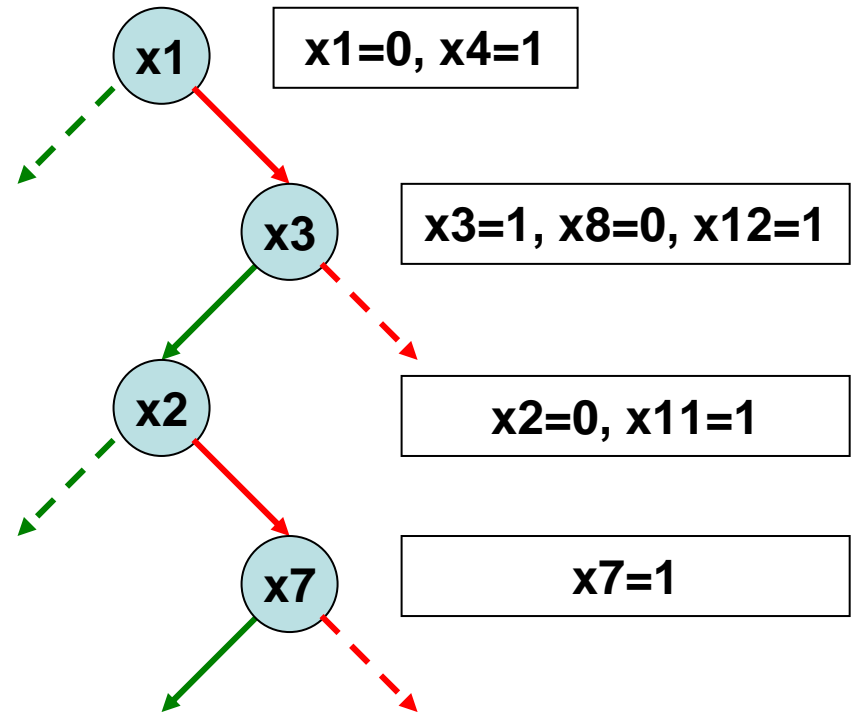
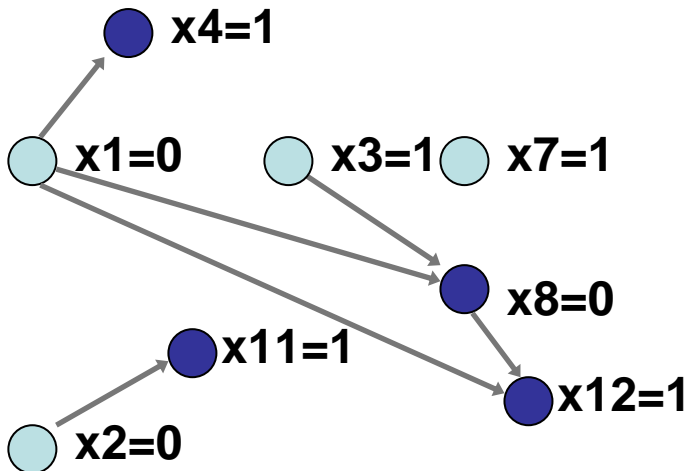
$x_2 + x_{11}$

$x_7' + x_3' + x_9$

$x_7' + x_8 + x_9'$

$x_7 + x_8 + x_{10}'$

$x_7 + x_{10} + x_{12}'$



Conflict Driven Learning and Non-chronological Backtracking

$x_1 + x_4$

$x_1 + x_3' + x_8'$

$x_1 + x_8 + x_{12}$

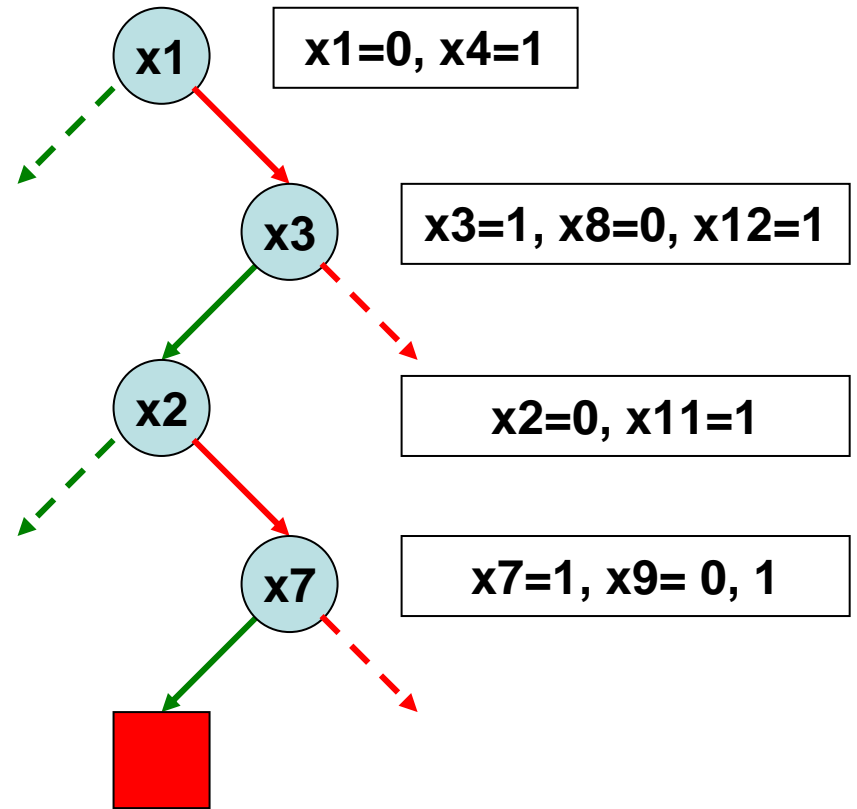
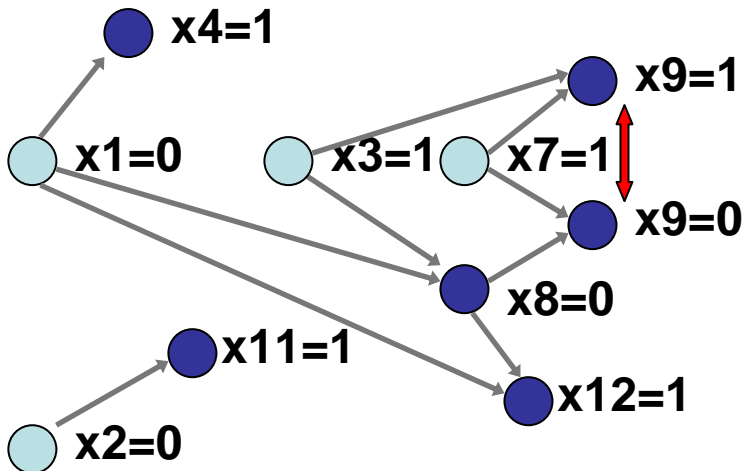
$x_2 + x_{11}$

$x_7' + x_3' + x_9$

$x_7' + x_8 + x_9'$

$x_7 + x_8 + x_{10}'$

$x_7 + x_{10} + x_{12}'$



Conflict Driven Learning and Non-chronological Backtracking

$x_1 + x_4$

$x_1 + x_3' + x_8'$

$x_1 + x_8 + x_{12}$

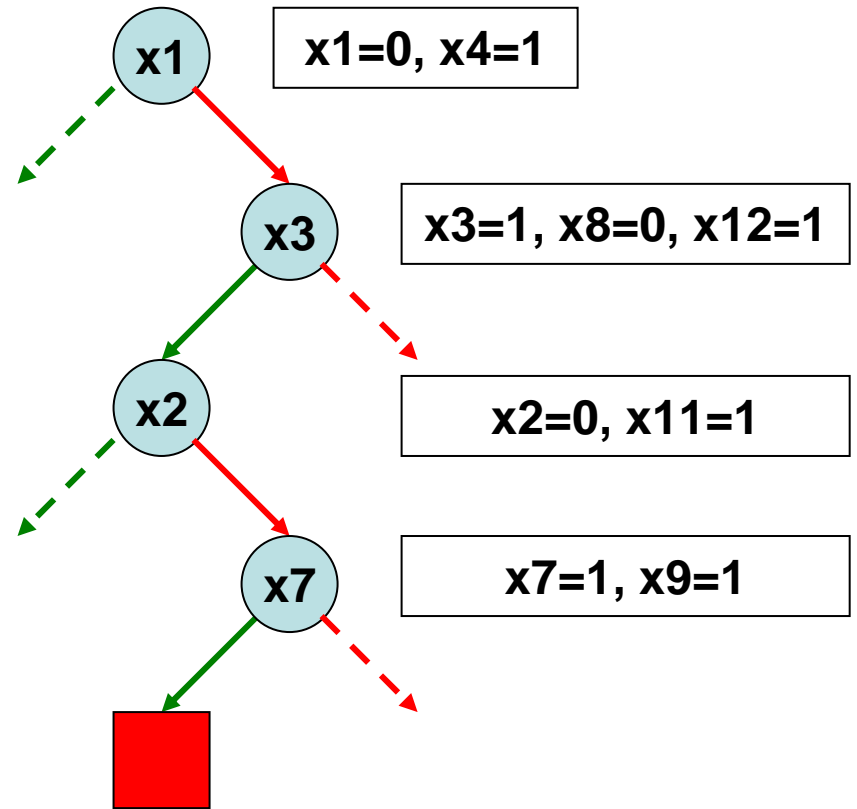
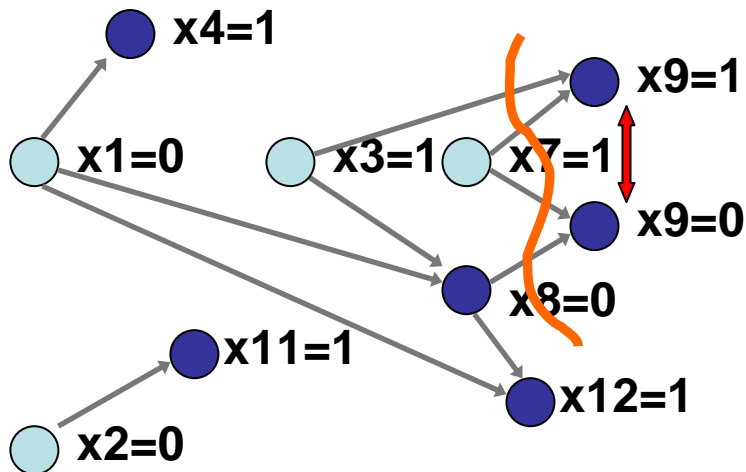
$x_2 + x_{11}$

$x_7' + x_3' + x_9$

$x_7' + x_8 + x_9'$

$x_7 + x_8 + x_{10}'$

$x_7 + x_{10} + x_{12}'$



$x_3=1 \wedge x_7=1 \wedge x_8=0 \rightarrow \text{conflict}$

Conflict Driven Learning and Non-chronological Backtracking

$x_1 + x_4$

$x_1 + x_3' + x_8'$

$x_1 + x_8 + x_{12}$

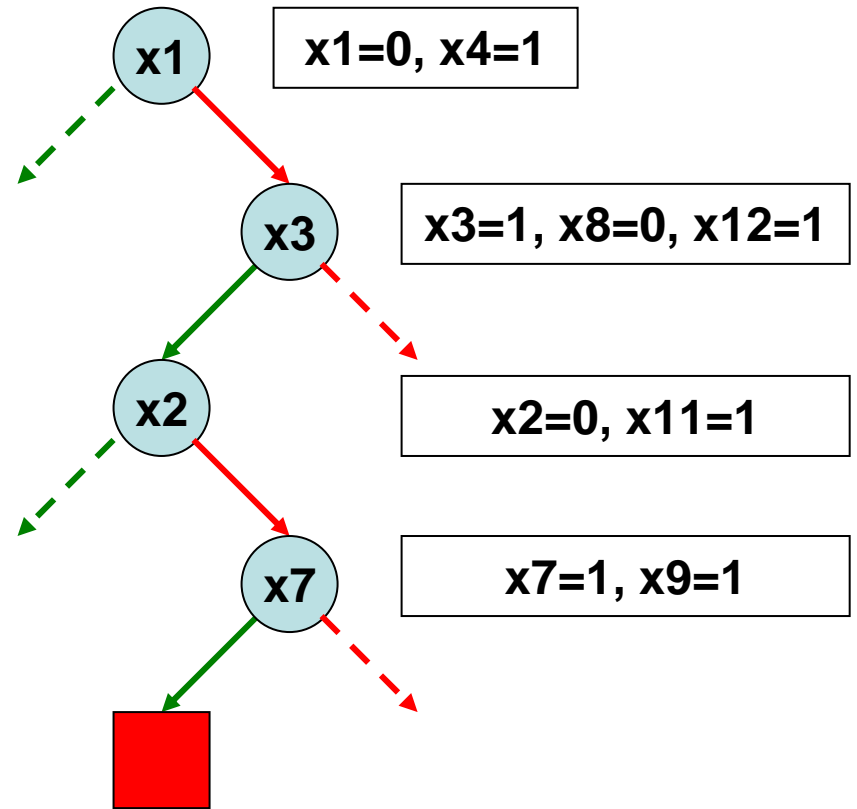
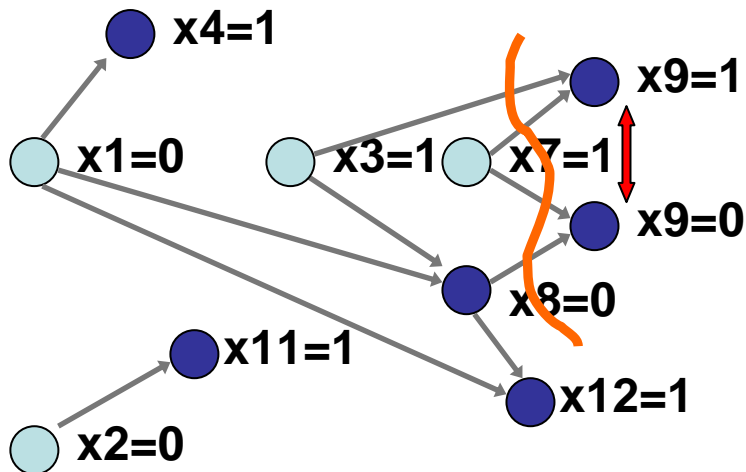
$x_2 + x_{11}$

$x_7' + x_3' + x_9$

$x_7' + x_8 + x_9'$

$x_7 + x_8 + x_{10}'$

$x_7 + x_{10} + x_{12}'$



$x_3=1 \wedge x_7=1 \wedge x_8=0 \rightarrow \text{conflict}$

Add conflict clause: $x_3' + x_7' + x_8$

Conflict Driven Learning and Non-chronological Backtracking

$x_1 + x_4$

$x_1 + x_3' + x_8'$

$x_1 + x_8 + x_{12}$

$x_2 + x_{11}$

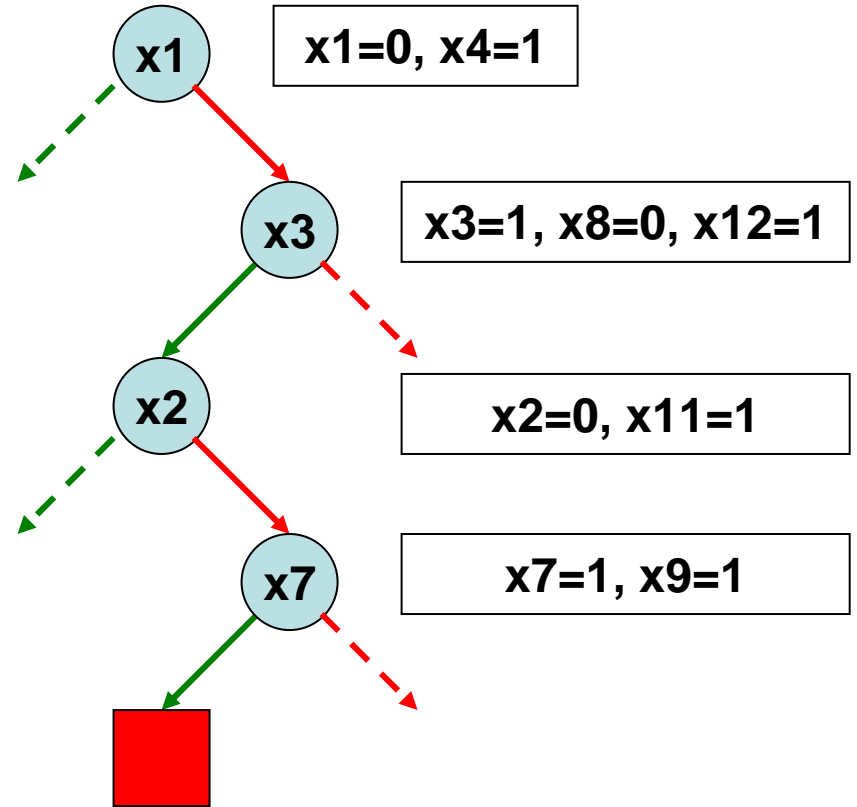
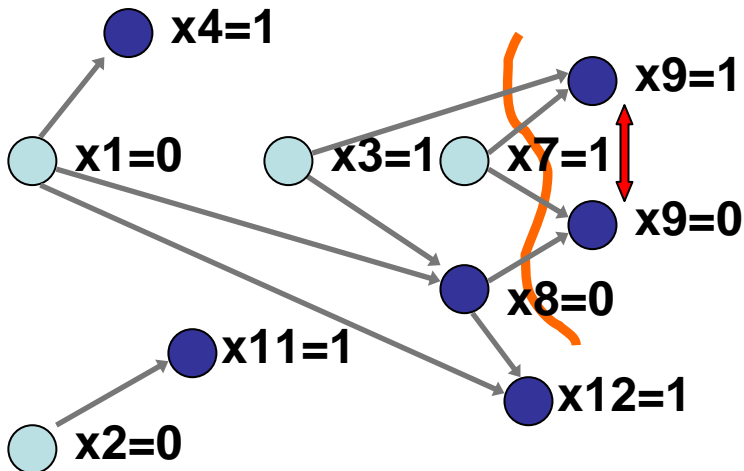
$x_7' + x_3' + x_9$

$x_7' + x_8 + x_9'$

$x_7 + x_8 + x_{10}'$

$x_7 + x_{10} + x_{12}'$

$x_3' + x_7' + x_8$



$x_3=1 \wedge x_7=1 \wedge x_8=0 \rightarrow \text{conflict}$

Add conflict clause: $x_3' + x_7' + x_8$

Conflict Driven Learning and Non-chronological Backtracking

$x_1 + x_4$

$x_1 + x_3' + x_8'$

$x_1 + x_8 + x_{12}$

$x_2 + x_{11}$

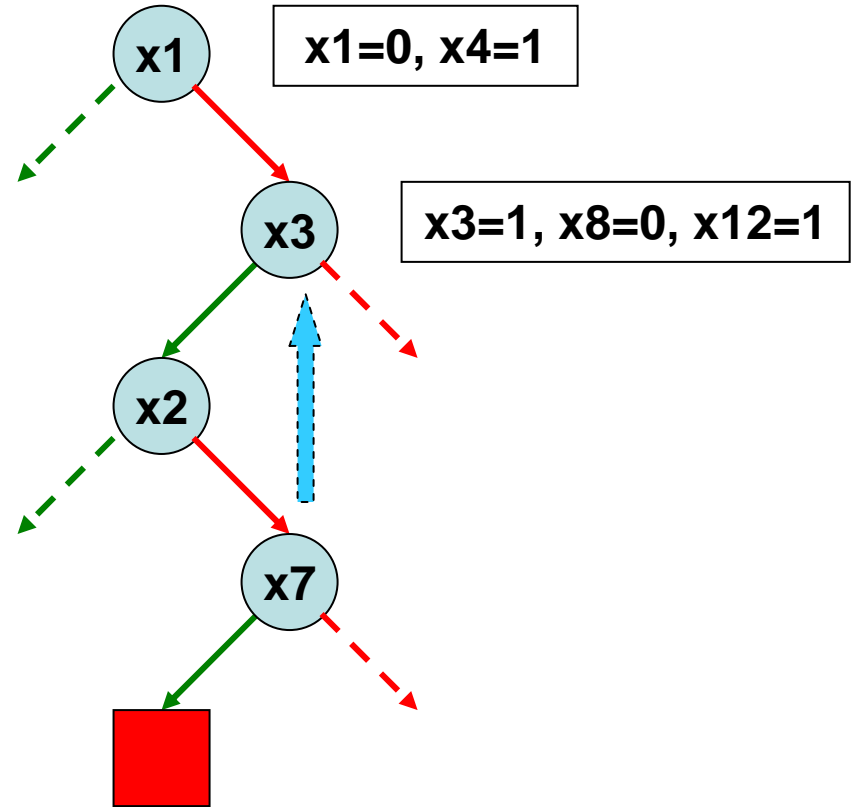
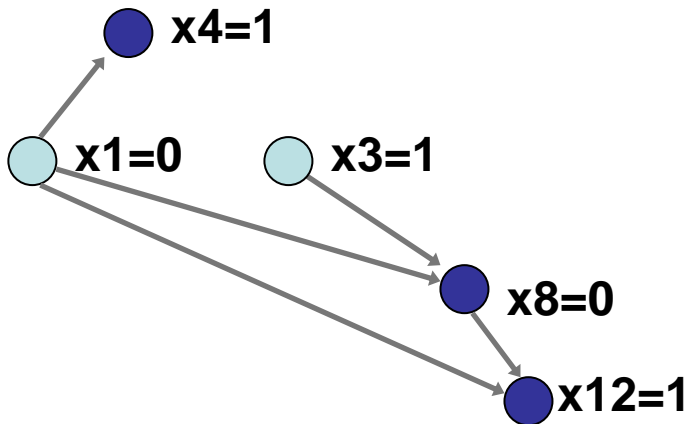
$x_7' + x_3' + x_9$

$x_7' + x_8 + x_9'$

$x_7 + x_8 + x_{10}'$

$x_7 + x_{10} + x_{12}'$

$x_3' + x_8 + x_7'$



Backtrack to the decision level of $x_3=1$
 $x_7 = 0$

Conflict Driven Learning and Non-chronological Backtracking

$x_1 + x_4$

$x_1 + x_3' + x_8'$

$x_1 + x_8 + x_{12}$

$x_2 + x_{11}$

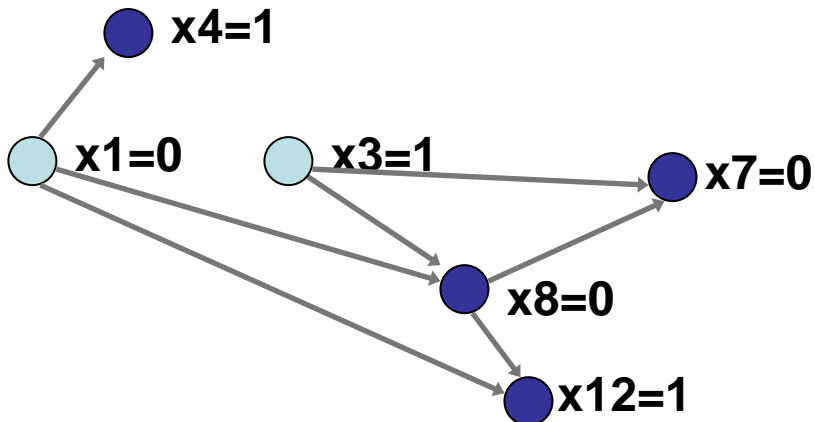
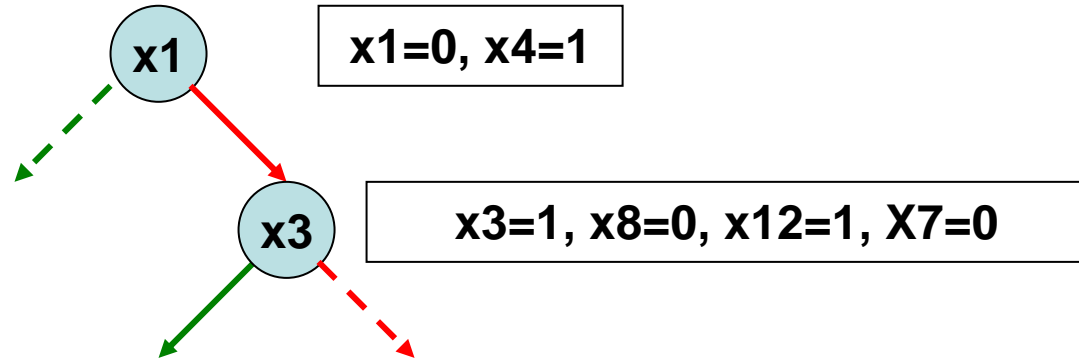
$x_7' + x_3' + x_9$

$x_7' + x_8 + x_9'$

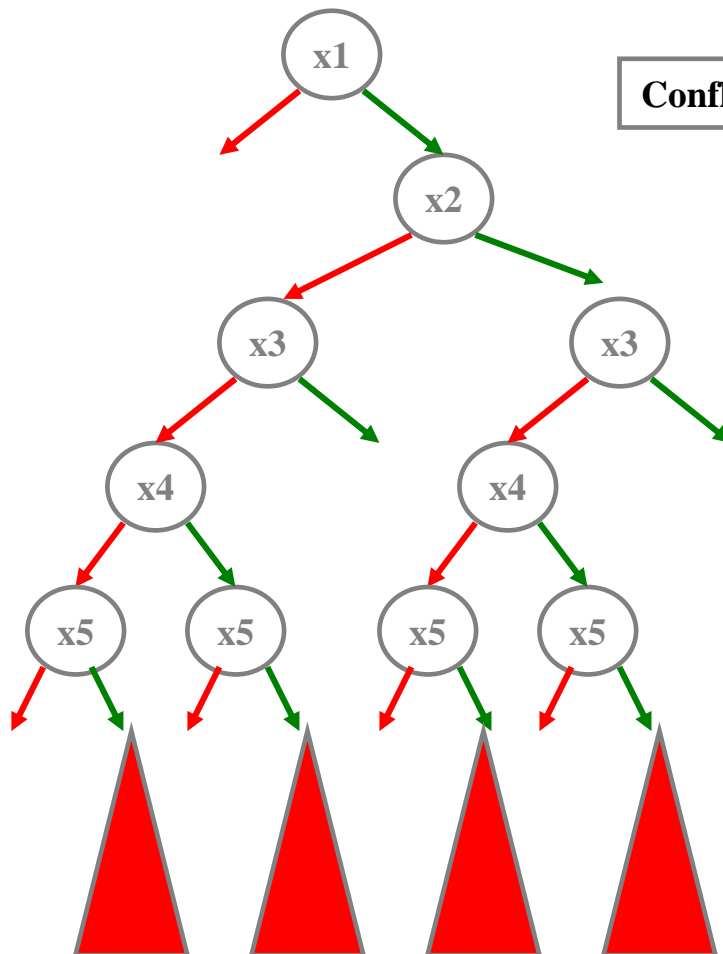
$x_7 + x_8 + x_{10}'$

$x_7 + x_{10} + x_{12}'$

$x_3' + x_8 + x_7'$



What's the big deal?

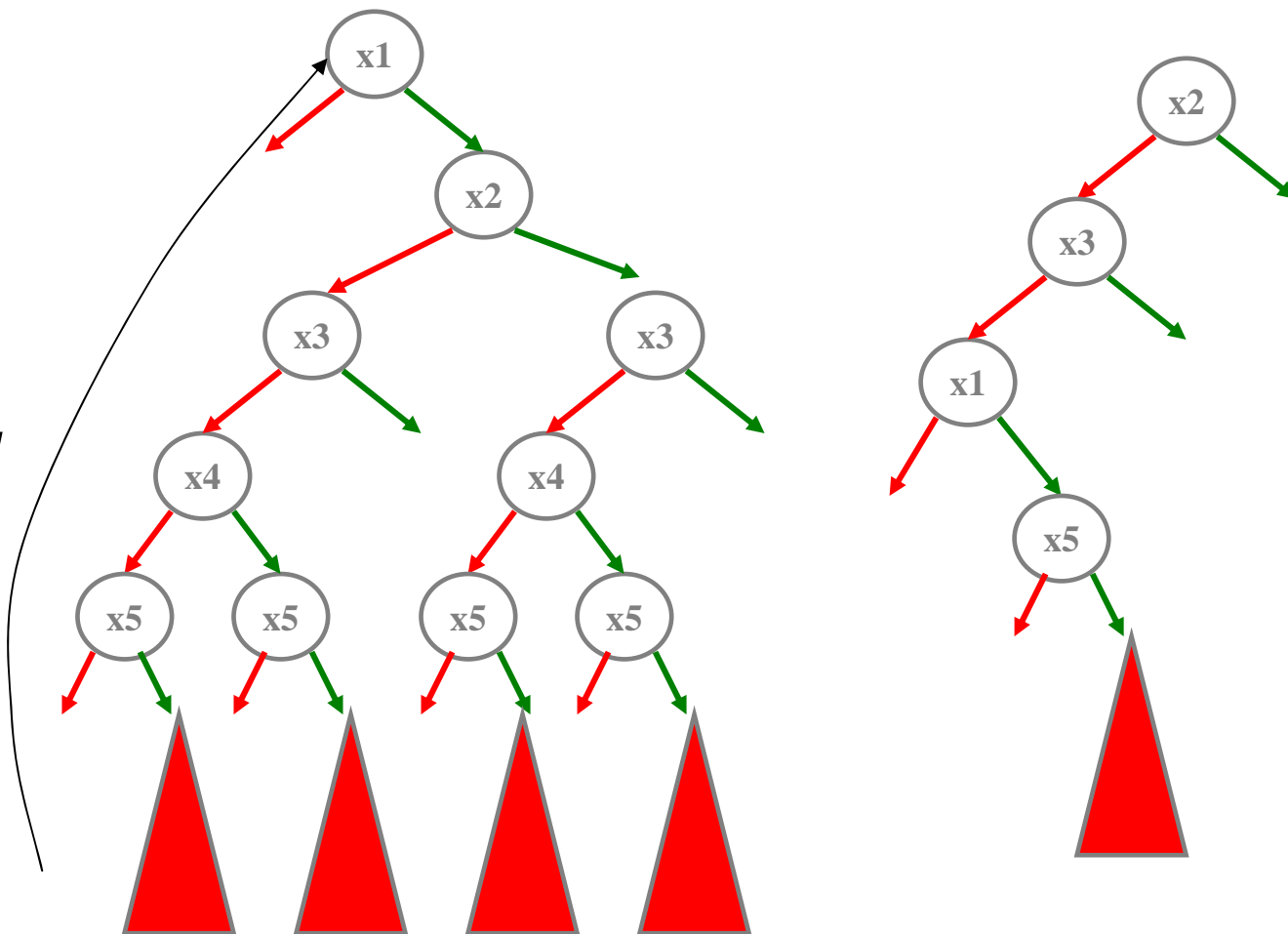


Significantly prune the search space –
learned clause is useful forever!

Useful in generating future conflict
clauses.

Restart

- Abandon the current search tree and reconstruct a new one
- The clauses learned prior to the restart are *still there* after the restart and can help pruning the search space
- Adds to robustness in the solver



Conflict clause: $x1' + x3 + x5'$

SAT becomes practical!

- Conflict driven learning greatly increases the capacity of SAT solvers (several thousand variables) for structured problems
- Realistic applications become feasible
 - Usually thousands and even millions of variables
 - Typical EDA applications that can make use of SAT
 - circuit verification
 - FPGA routing
 - many other applications...
- Research direction changes towards more efficient implementations

Large Example: Tough

- Industrial Processor Verification
 - Bounded Model Checking, 14 cycle behavior
- Statistics
 - 1 million variables
 - 10 million literals initially
 - 200 million literals including added clauses
 - 30 million literals finally
 - 4 million clauses (initially)
 - 200K clauses added
 - 1.5 million decisions
 - 3 hours run time

Chaff

- One to two orders of magnitude faster than other solvers...
 - M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, “Chaff: Engineering an Efficient SAT Solver” *Proc. DAC* 2001.
- Widely Used:
 - BlackBox – AI Planning
 - Henry Kautz (UW)
 - NuSMV – Symbolic Verification toolset
 - A. Cimatti, et. al. “NuSMV 2: An Open Source Tool for Symbolic Model Checking” *Proc. CAV* 2002.
 - GrAnDe – Automatic theorem prover
 - Several industrial licenses

Chaff Philosophy

- Make the core operations fast
 - profiling driven, most time-consuming parts:
 - Boolean Constraint Propagation (BCP) and Decision
- Emphasis on coding efficiency and elegance
- Emphasis on optimizing data cache behavior
- As always, good search space pruning (i.e. conflict resolution and learning) is important

Motivating Metrics: Decisions, Instructions, Cache Performance and Run Time

	1dlx_c_mc_ex_bp_f
Num Variables	776
Num Clauses	3725
Num Literals	10045

	Z-Chaff	GRASP
# Decisions	3166	1795
# Instructions	86.6M	1415.9M
# L1/L2 accesses	24M / 1.7M	416M / 153M
% L1/L2 misses	4.8% / 4.6%	32.9% / 50.3%
# Seconds	0.22	11.78

BCP Algorithm

- What “causes” an implication? When can it occur?
 - All literals in a clause but one are assigned to F
 - $(v_1 + v_2 + v_3)$: implied cases: $(0 + 0 + v_3)$ or $(0 + v_2 + 0)$ or $(v_1 + 0 + 0)$
 - For an N-literal clause, this can only occur after N-1 of the literals have been assigned to F
 - So, (theoretically) we could completely ignore the first N-2 assignments to this clause
 - In reality, we pick two literals in each clause to “watch” and thus can ignore any assignments to the other literals in the clause.
 - Example: $(v_1 + v_2 + v_3 + v_4 + v_5)$
 - $(v_1=X + v_2=X + v_3=? \text{ {i.e. X or 0 or 1}} + v_4=? + v_5=?)$

BCP Algorithm

- Big Invariants
 - Each clause has two watched literals
 - If a clause can become newly implied via any sequence of assignments, then this sequence will include an assignment of one of the watched literals to F.
 - Example again: $(v1 + v2 + v3 + v4 + v5)$
 - $(\mathbf{v1=X} + \mathbf{v2=X} + v3=? + v4=? + v5=?)$
- BCP consists of identifying implied clauses (and the associated implications) while maintaining the “Big Invariants”
- No actions on backtracking

BCP Algorithm

- Let's illustrate this with an example:

$v_2 + v_3 + v_1 + v_4 + v_5$

$v_1 + v_2 + v_3'$

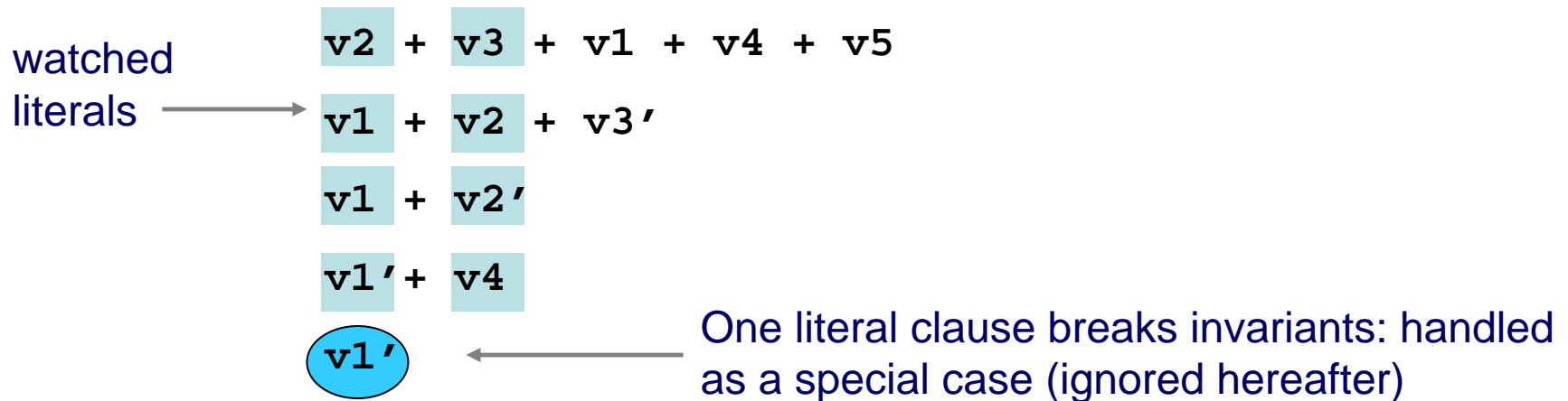
$v_1 + v_2'$

$v_1' + v_4$

v_1'

BCP Algorithm

- Let's illustrate this with an example:



- Initially, we identify any two literals in each clause as the watched ones
- Clauses of size one are a special case

BCP Algorithm

- We begin by processing the assignment $v_1 = F$ (which is implied by the size one clause)

State: ($v_1 = F$)

Pending:

$$v_2 + v_3 + v_1 + v_4 + v_5$$

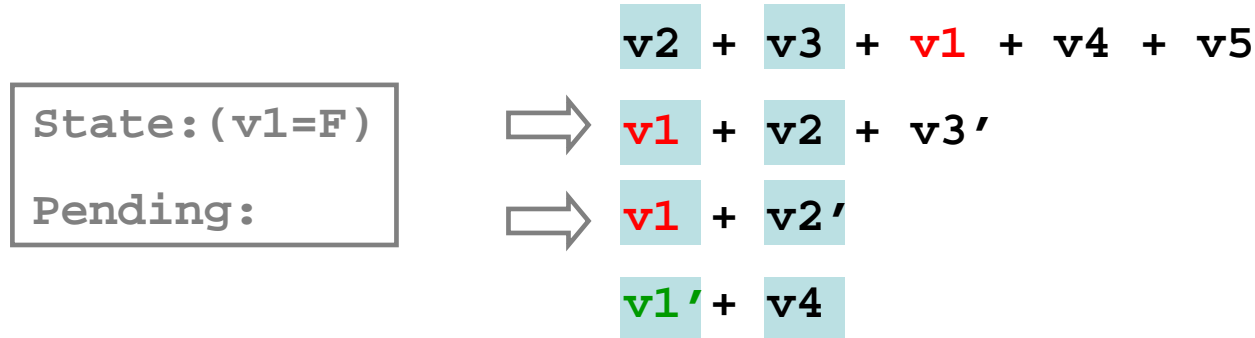
$$v_1 + v_2 + v_3'$$

$$v_1 + v_2'$$

$$v_1' + v_4$$

BCP Algorithm

- We begin by processing the assignment $v_1 = F$ (which is implied by the size one clause)



- To maintain our invariants, we must examine each clause where the assignment being processed has set a watched literal to F

BCP Algorithm

- We begin by processing the assignment $v_1 = F$ (which is implied by the size one clause)

State: ($v_1 = F$)
Pending:

$$v_2 + v_3 + v_1 + v_4 + v_5$$

$$v_1 + v_2 + v_3'$$

$$v_1 + v_2'$$

$$\Rightarrow v_1' + v_4$$

- To maintain our invariants, we must examine each clause where the assignment being processed has set a watched literal to F.
- We need not process clauses where a watched literal has been set to T, because the clause is now satisfied and so can not become implied.

BCP Algorithm

- We begin by processing the assignment $v_1 = F$ (which is implied by the size one clause)

State: ($v_1 = F$)

Pending:

$$\Rightarrow v_2 + v_3 + v_1 + v_4 + v_5$$
$$v_1 + v_2 + v_3'$$
$$v_1 + v_2'$$
$$v_1' + v_4$$

- To maintain our invariants, we must examine each clause where the assignment being processed has set a watched literal to F.
- We need not process clauses where a watched literal has been set to T, because the clause is now satisfied and so can not become implied.
- We *certainly* need not process any clauses where neither watched literal changes state (in this example, where v_1 is not watched).

BCP Algorithm

- Now let's actually process the second and third clauses:

$$v_2 + v_3 + v_1 + v_4 + v_5$$

$$v_1 + v_2 + v_3'$$

$$v_1 + v_2'$$

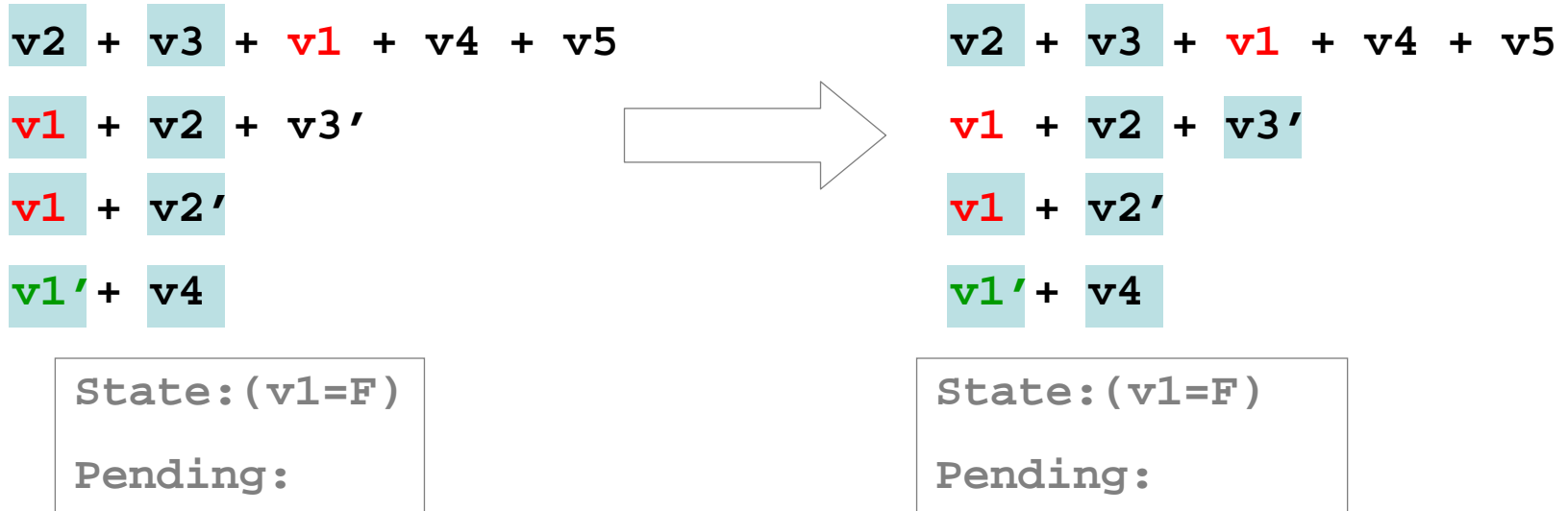
$$v_1' + v_4$$

State: (v1=F)

Pending:

BCP Algorithm

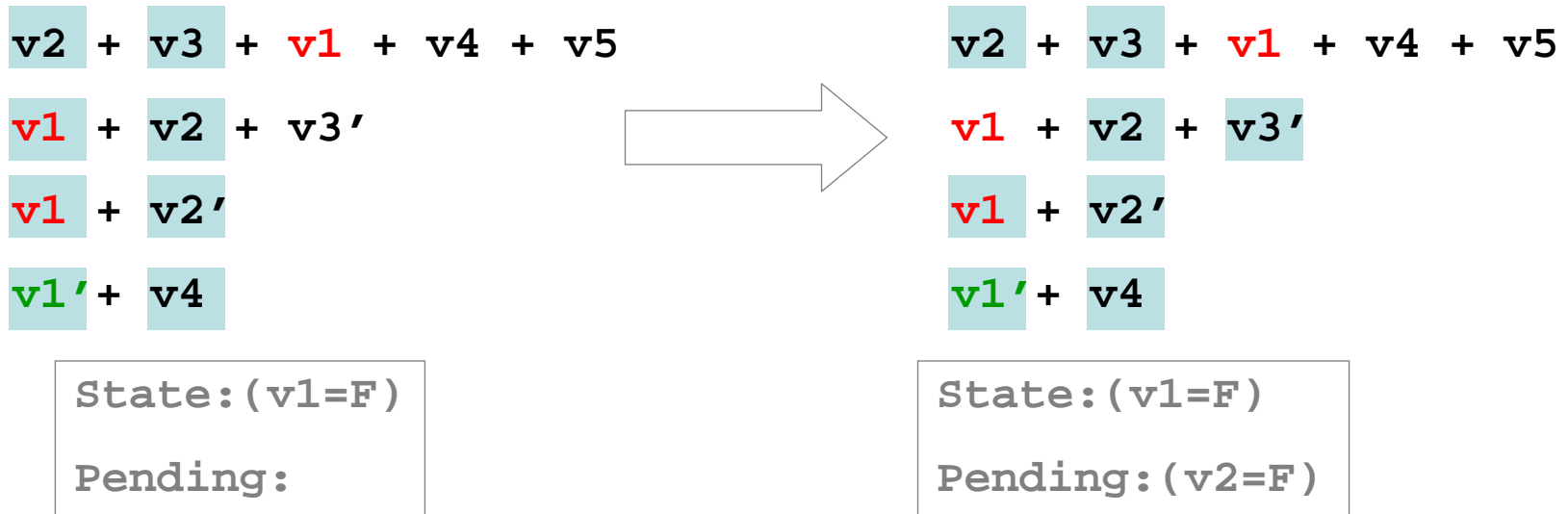
- Now let's actually process the second and third clauses:



- For the second clause, we replace $v1$ with $v3'$ as a new watched literal. Since $v3'$ is not assigned to F, this maintains our invariants.

BCP Algorithm

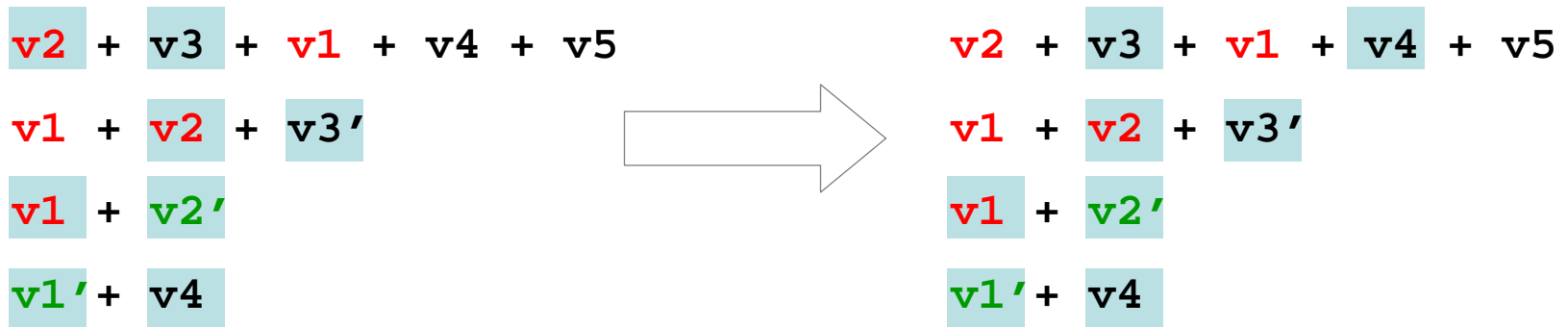
- Now let's actually process the second and third clauses:



- For the second clause, we replace $v1$ with $v3'$ as a new watched literal. Since $v3'$ is not assigned to F, this maintains our invariants.
- The third clause is implied. We record the new implication of $v2'$, and add it to the queue of assignments to process. Since the clause cannot again become newly implied, our invariants are maintained.

BCP Algorithm

- Next, we process $v2'$. We only examine the first 2 clauses.



State: $(v1=F, v2=F)$

Pending:

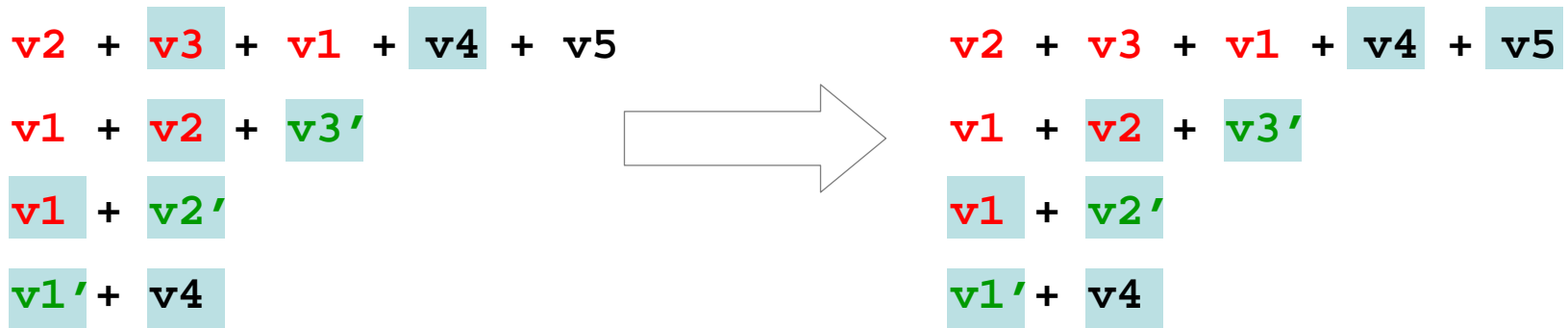
State: $(v1=F, v2=F)$

Pending: $(v3=F)$

- For the first clause, we replace $v2$ with $v4$ as a new watched literal. Since $v4$ is not assigned to F , this maintains our invariants.
- The second clause is implied. We record the new implication of $v3'$, and add it to the queue of assignments to process. Since the clause cannot again become newly implied, our invariants are maintained.

BCP Algorithm

- Next, we process $v3'$. We only examine the first clause.



State: $(v1=F, v2=F, v3=F)$

Pending:

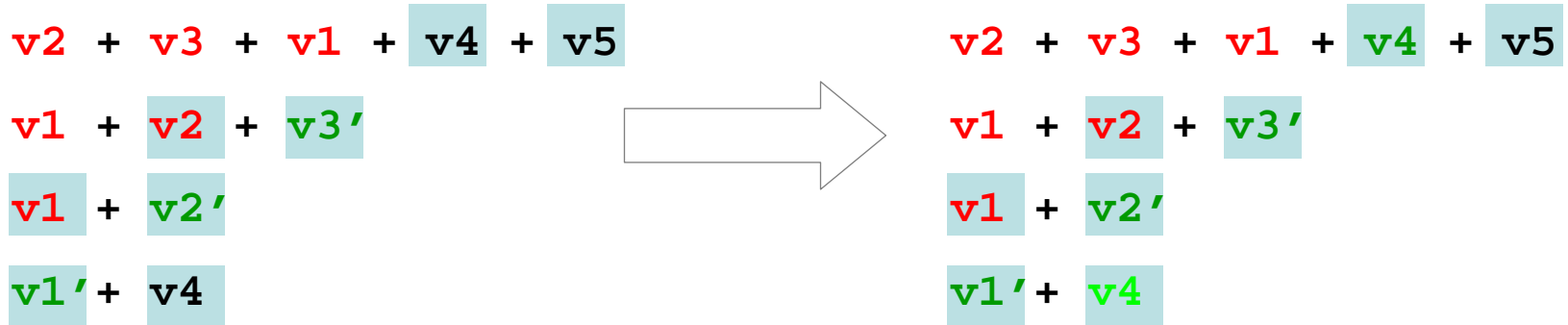
State: $(v1=F, v2=F, v3=F)$

Pending:

- For the first clause, we replace $v3$ with $v5$ as a new watched literal. Since $v5$ is not assigned to F , this maintains our invariants.
- Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. Both $v4$ and $v5$ are unassigned. Let's say we decide to assign $v4=T$ and proceed.

BCP Algorithm

- Next, we process v_4 . We do nothing at all.



State: ($v_1=F$, $v_2=F$, $v_3=F$,
 $v_4=T$)

State: ($v_1=F$, $v_2=F$, $v_3=F$,
 $v_4=T$)

- Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. Only v_5 is unassigned. Let's say we decide to assign $v_5=F$ and proceed.

BCP Algorithm

- Next, we process $v_5=F$. We examine the first clause.

$$\begin{array}{l} v_2 + v_3 + v_1 + v_4 + v_5 \\ v_1 + v_2 + v_3' \\ v_1 + v_2' \\ v_1' + v_4 \end{array} \quad \longrightarrow \quad \begin{array}{l} v_2 + v_3 + v_1 + v_4 + v_5 \\ v_1 + v_2 + v_3' \\ v_1 + v_2' \\ v_1' + v_4 \end{array}$$

State: ($v_1=F$, $v_2=F$, $v_3=F$,
 $v_4=T$, $v_5=F$)

State: ($v_1=F$, $v_2=F$, $v_3=F$,
 $v_4=T$, $v_5=F$)

- The first clause is implied. However, the implication is $v_4=T$, which is a duplicate (since $v_4=T$ already) so we ignore it
- Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. No variables are unassigned, so the problem is sat, and we are done

BCP Algorithm Summary

- During forward progress: Decisions and Implications
 - Only need to examine clauses where watched literal is set to F
 - Can ignore any assignments of literals to T
 - Can ignore any assignments to non-watched literals
- During backtrack: Unwind Assignment Stack
 - Any sequence of chronological unassignments will maintain our invariants
 - So no action is required at all to unassign variables
- Overall
 - Minimize clause access
 - Better memory locality

Decision Heuristics – Conventional Wisdom

- DLIS is a relatively simple dynamic decision heuristic
 - Simple and intuitive: At each decision simply choose the assignment that satisfies the most unsatisfied clauses
 - However, considerable work is required to maintain the statistics necessary for this heuristic – for one implementation:
 - Must touch *every* clause that contains a literal that has been set to true. Often restricted to initial (not learned) clauses
 - Maintain “sat” counters for each clause
 - When counters transition $0 \rightarrow 1$, update rankings
 - Need to reverse the process for unassignment
 - The total effort required for this and similar decision heuristics is *much more* than for our BCP algorithm.
- Look ahead algorithms even more compute intensive
 - C. Li, Anbulagan, “Look-ahead versus look-back for satisfiability problems” *Proc. of CP*, 1997.

Chaff Decision Heuristic - VSIDS

- Variable State Independent Decaying Sum
 - Rank variables by literal count in the initial clause database
 - Periodically, divide all counts by a constant
 - Only increment counts as new clauses are added
- Quasi-static:
 - Static because it doesn't depend on var state
 - Not static because it gradually changes as new clauses are added
 - Decay causes bias toward *recent* conflicts.
- Use heap to find unassigned var with the highest ranking
 - Even single linear pass through variables on each decision would dominate run-time!
- Seems to work fairly well in terms of # decisions
 - hard to compare with other heuristics because they have too much overhead

Interplay of BCP and Decision

- This is only an intuitive description ...
 - Reality depends heavily on specific instance
- Take some variable ranking (from the decision engine)
 - Assume several decisions are made
 - Say $v_2=T$, $v_7=F$, $v_9=T$, $v_1=T$ (and any implications thereof)
 - Then a conflict is encountered that forces $v_2=F$
 - The next decisions may still be $v_7=F$, $v_9=T$, $v_1=T$!
 - But the BCP engine has recently processed these assignments ... so these variables are unlikely to still be watched.
 - Thus, the BCP engine *inherently does a differential update.*
 - And the Decision heuristic makes differential changes more likely to occur in practice.
- In a more general sense, the more “active” a variable is, the more likely it is to *not* be watched.

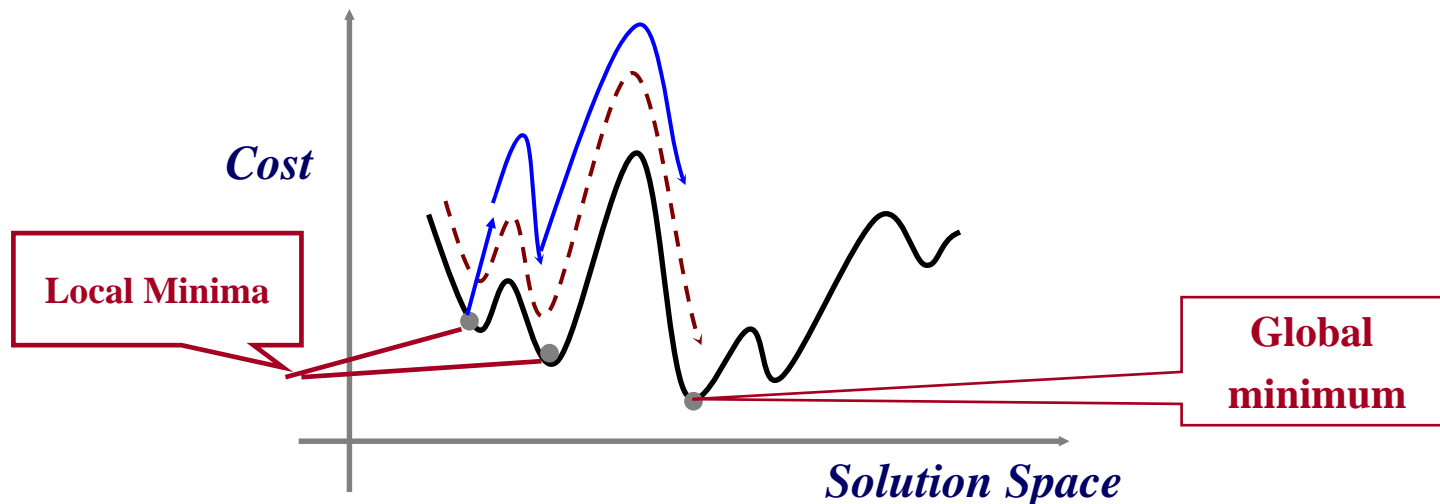
Missing

- Post Chaff SAT solvers
 - BerkMin
 - Seige
 - miniSat
 - HaifaSAT
 - JeruSAT (Alex Nadel)
- The Stålmarck's algorithm
- Hyperresolution
- Local Search

Local Search (GSAT, WSAT)

B. Selman, H. Levesque, and D. Mitchell. "A new method for solving hard satisfiability problems". *Proc. AAAI*, 1992.

- Incomplete SAT solvers
 - Geared towards satisfiable instances, cannot prove unsatisfiability
- Hill climbing algorithm for local search
- Make short local moves
- Probabilistically accept moves that worsen the cost function to enable exits from local minima



Bibliography

- **Chaff: Engineering an Efficient SAT Solver**
Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik (DAC'01)
- **Efficient Conflict Driven Learning in a Boolean Satisfiability Solver** Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz (IJCAD'01)
- **A New Method for Solving Hard Satisfiability Problems**
Bart Selman, Hector Levesque, David Mitchell (AAI'92)

Open Question

- Is there a subset of propositional logic beyond Horn clauses which:
 - Allows polynomial SAT
 - Includes many of the practical instances

Summary

- Rich history of emphasis on practical efficiency
- Need to account for computation cost in search space pruning
- Need to match algorithms with underlying processing system architectures
- Specific problem classes can benefit from specialized algorithms
 - Identification of problem classes?
 - Dynamically adapting heuristics?
- We barely understand the tip of the iceberg here
 - much room to learn and improve