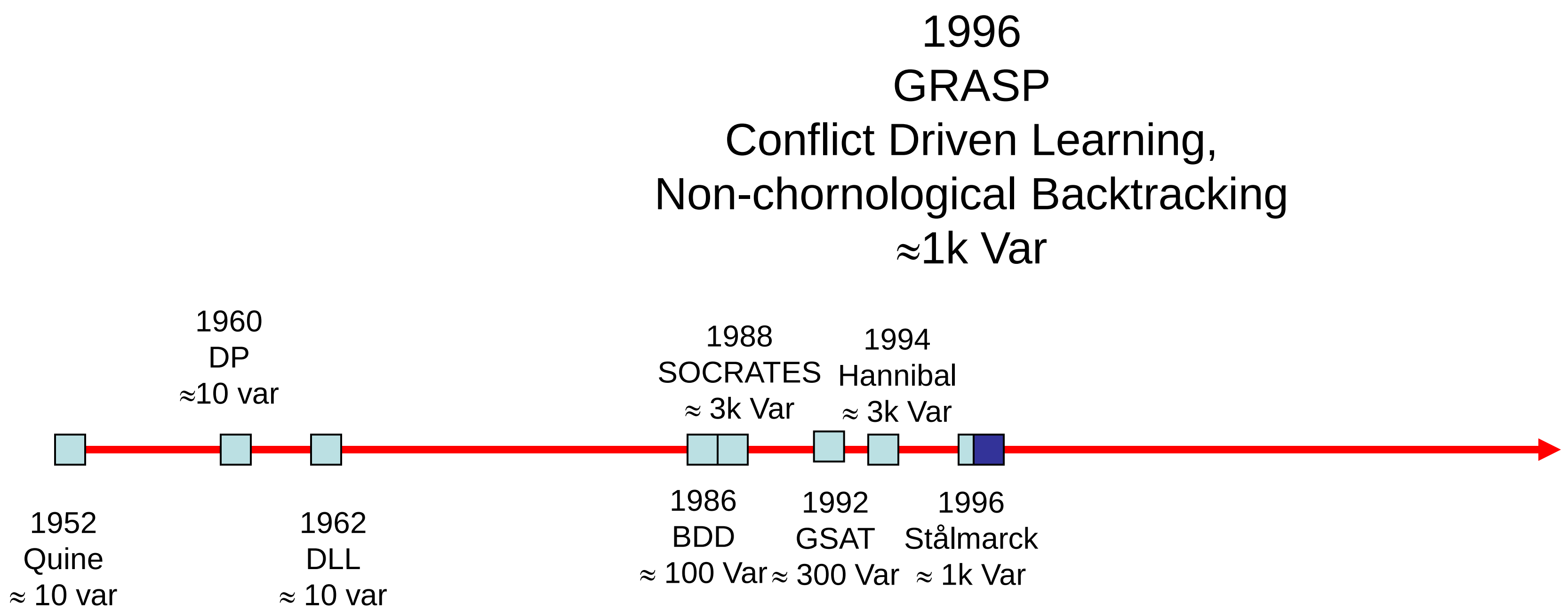
The Timeline



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.

GRASP

- Based on DPLL (backtracking) algorithm
- Key concepts: Conflict driven learning and non-chronological backtracking
- Bayardo and Schrag's RelSAT concurrently proposed conflict driven learning

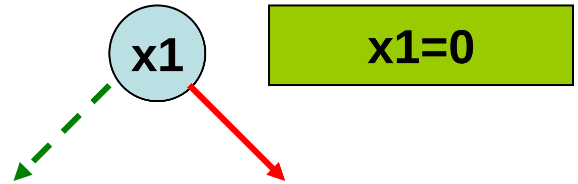
  R. J. Bayardo Jr. and R. C. Schrag "Using CSP look-back techniques to solve real world SAT instances." *Proc. AAAI*, pp. 203-208, 1997

Conflict Driven Learning

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

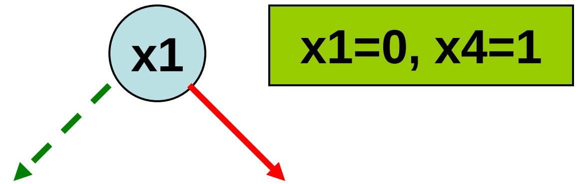
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

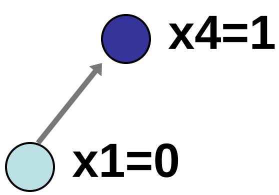


x1=0

Conflict Driven Learning

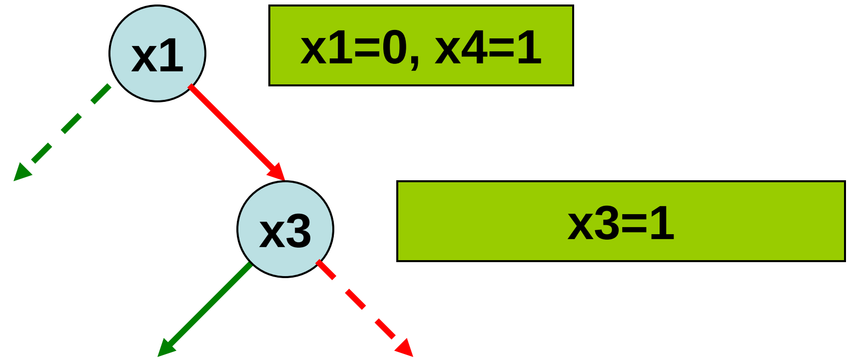
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

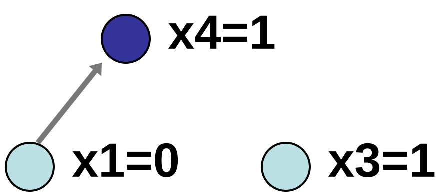




Conflict Driven Learning

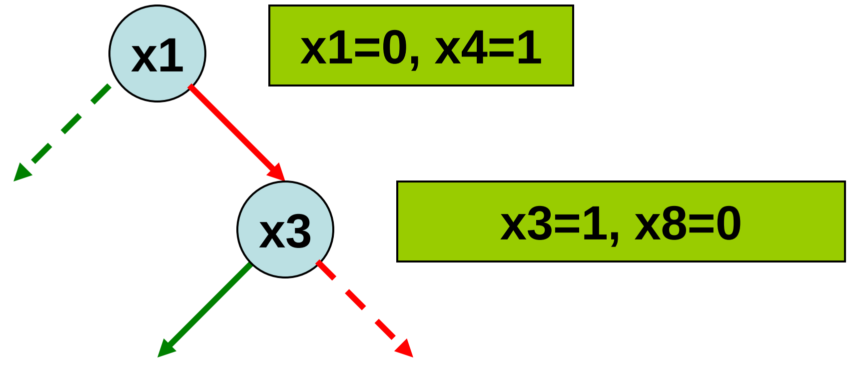
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

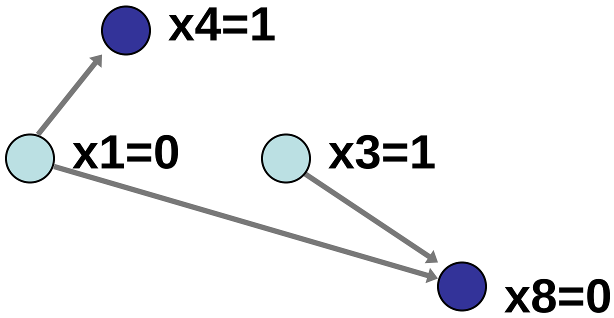




Conflict Driven Learning

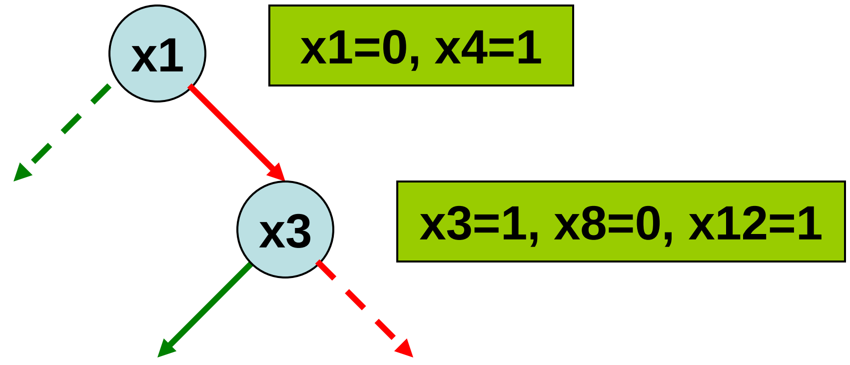
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

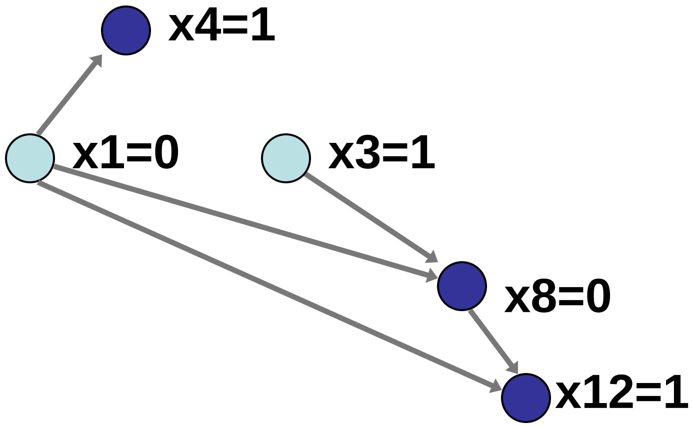




Conflict Driven Learning

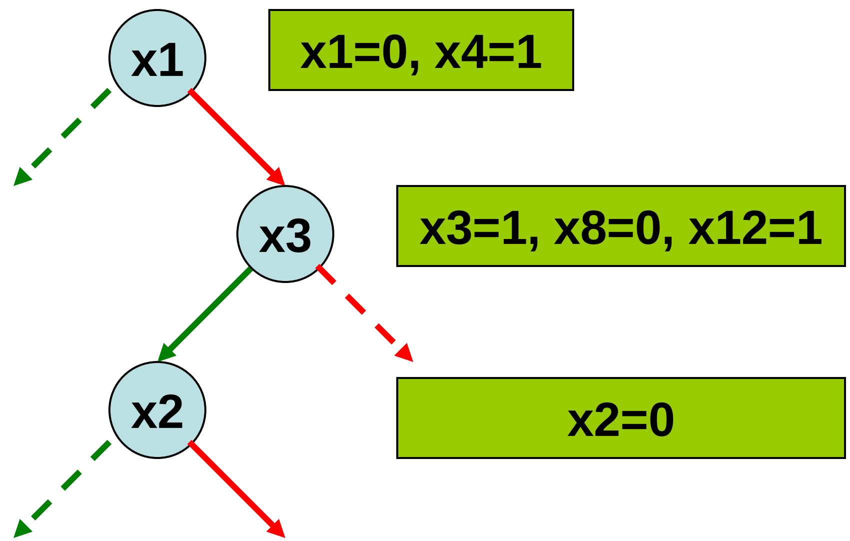
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

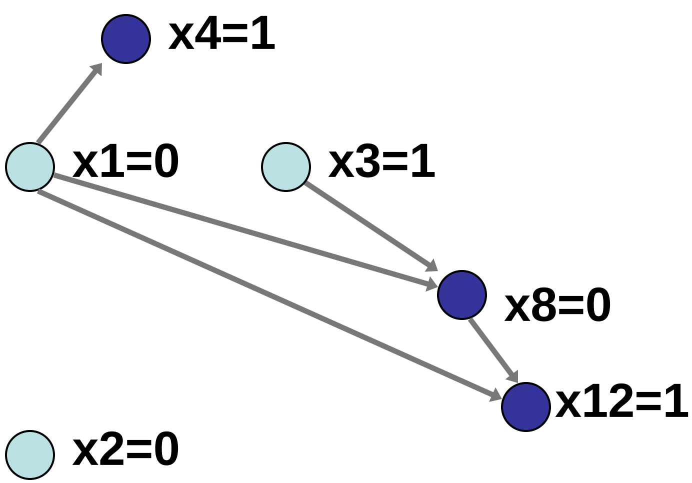




Conflict Driven Learning

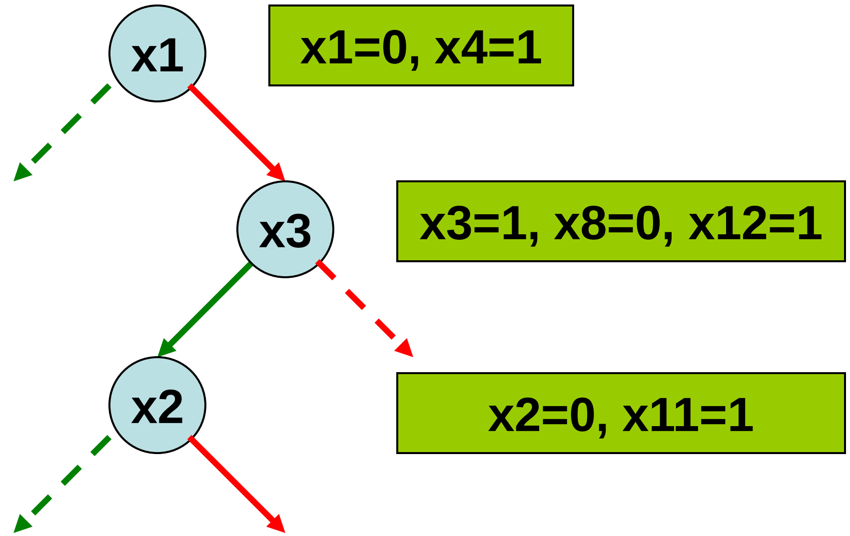
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

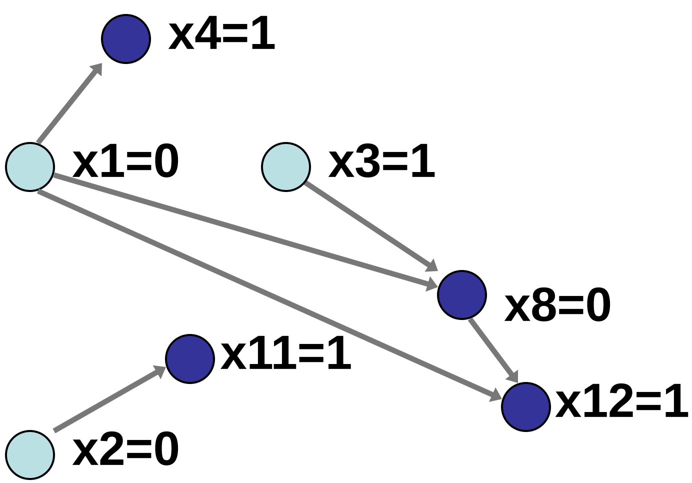




Conflict Driven Learning

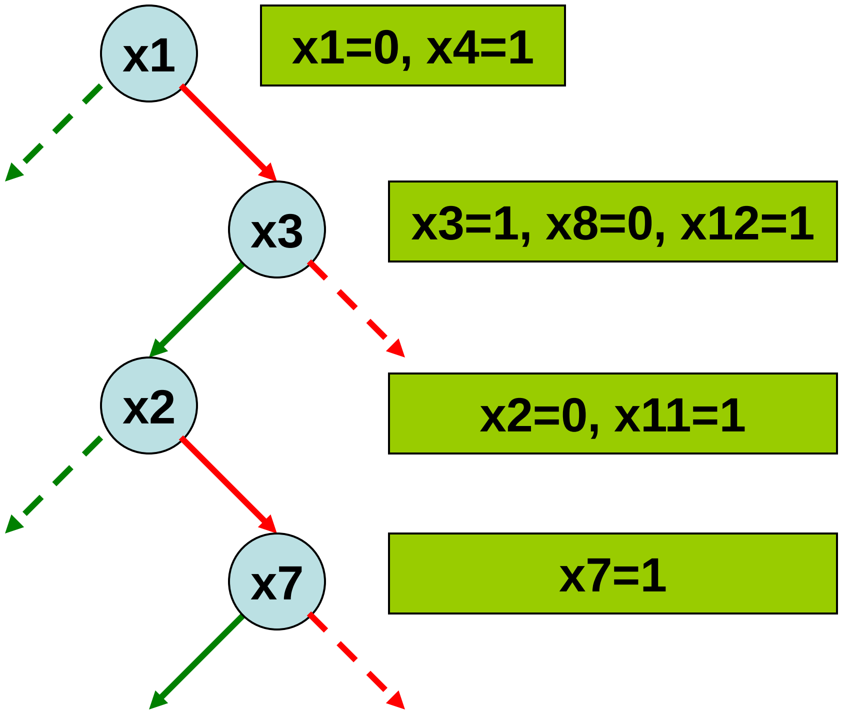
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

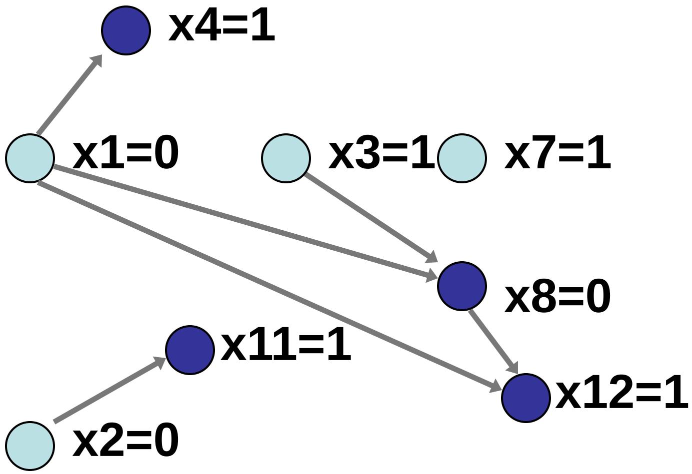




Conflict Driven Learning

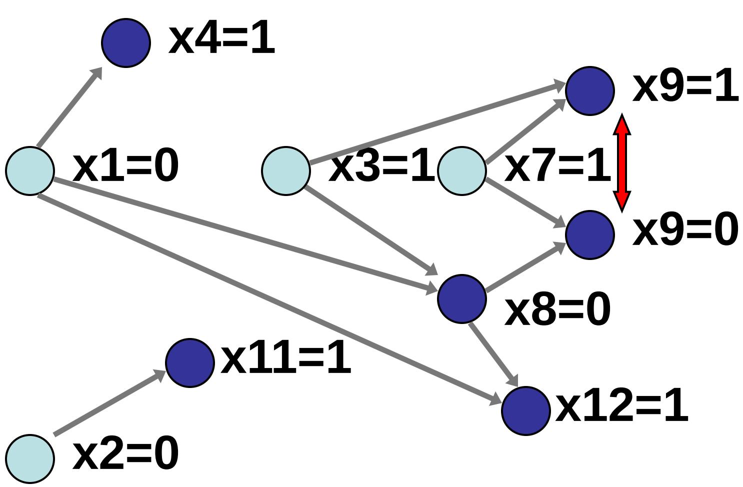
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

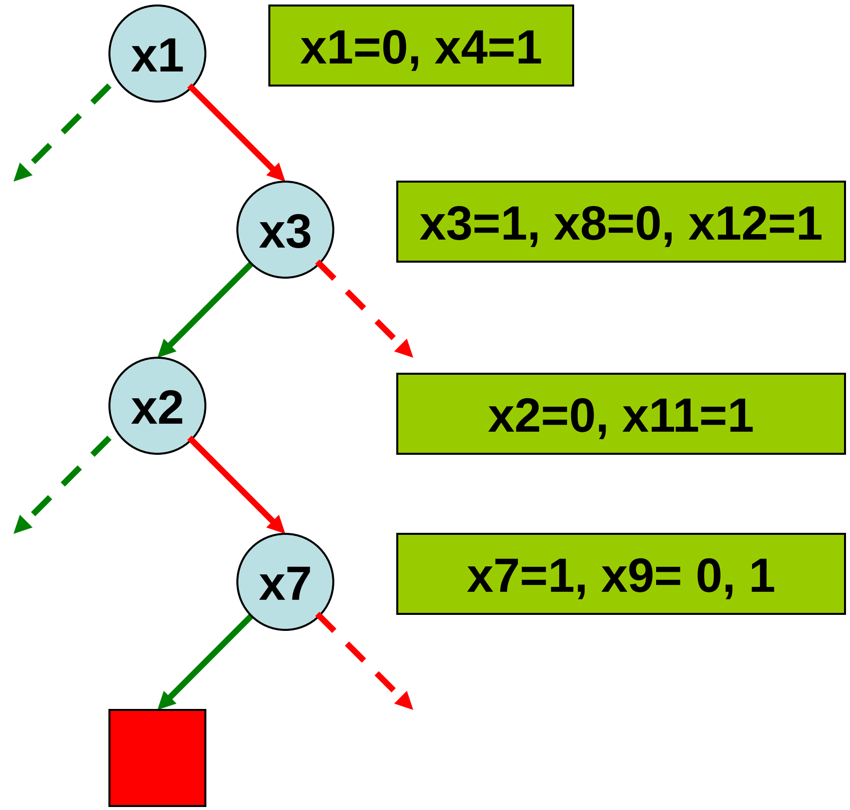




Conflict Driven Learning

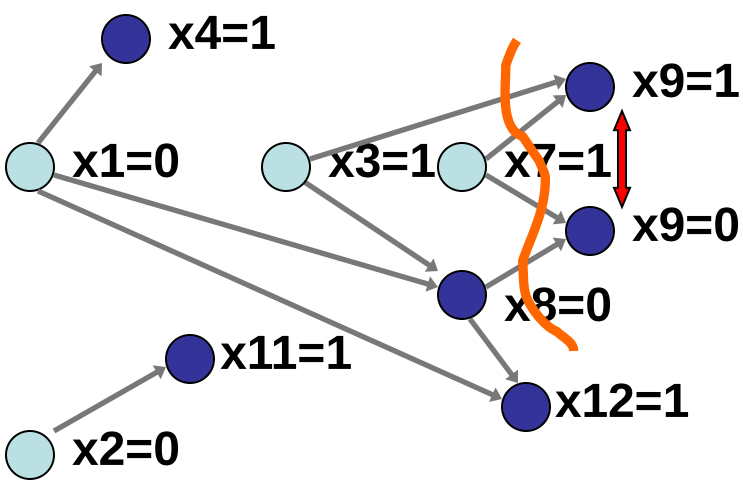
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

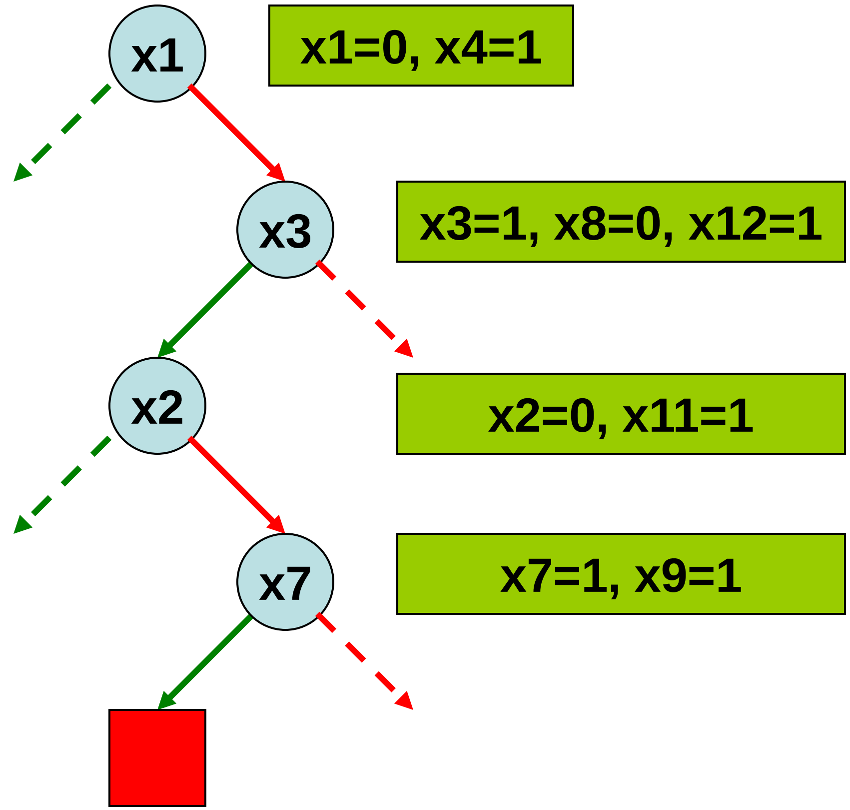




Conflict Driven Learning

x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

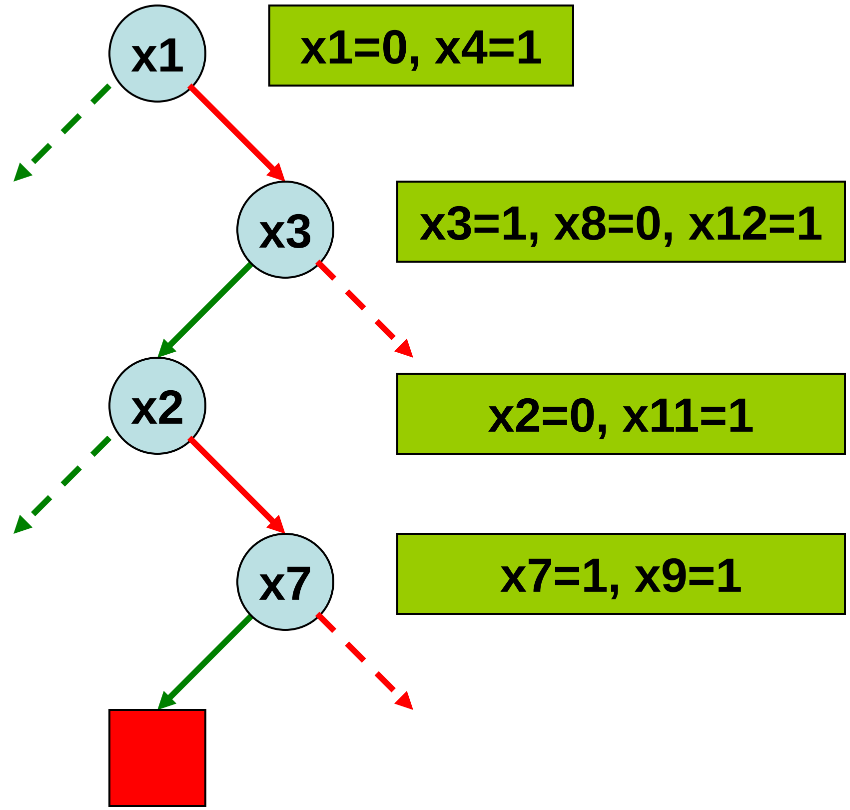


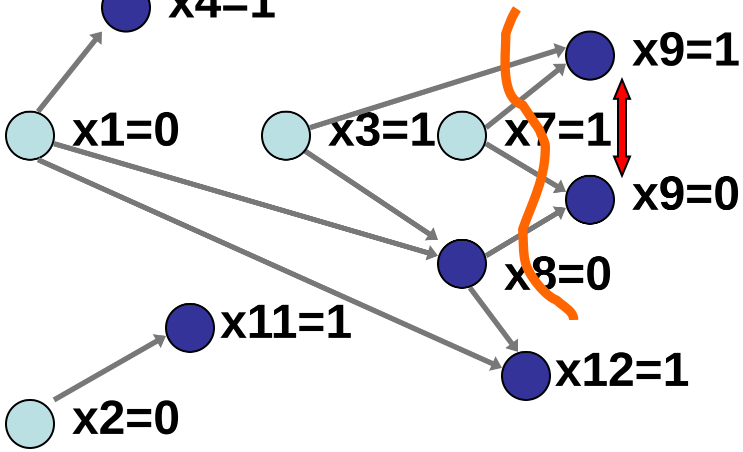


$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow$ conflict

Conflict Driven Learning

x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’





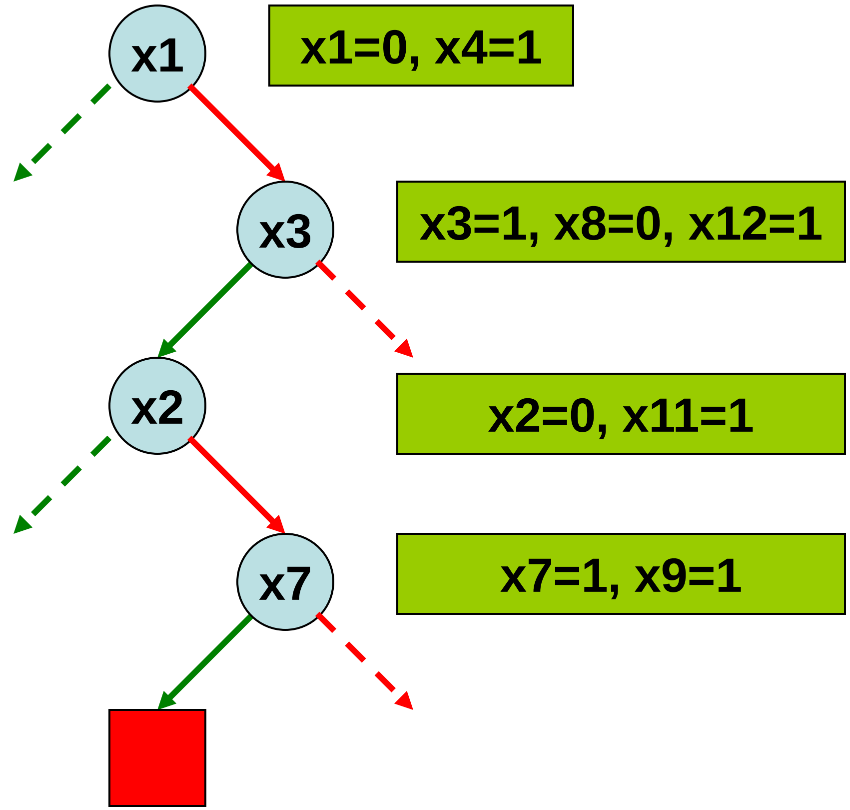
x3=1∧x7=1∧x8=0 → conflict
Add conflict clause: x3’+x7’+x8

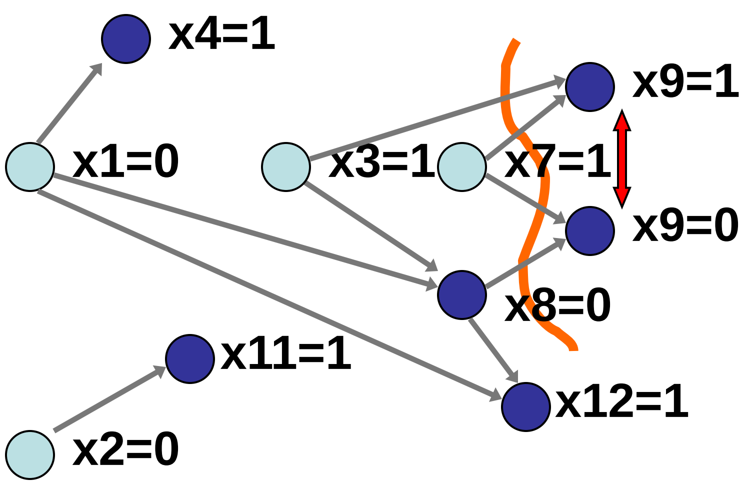
Conflict Driven Learning

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

Conflict clause

x3'+x7'+x8



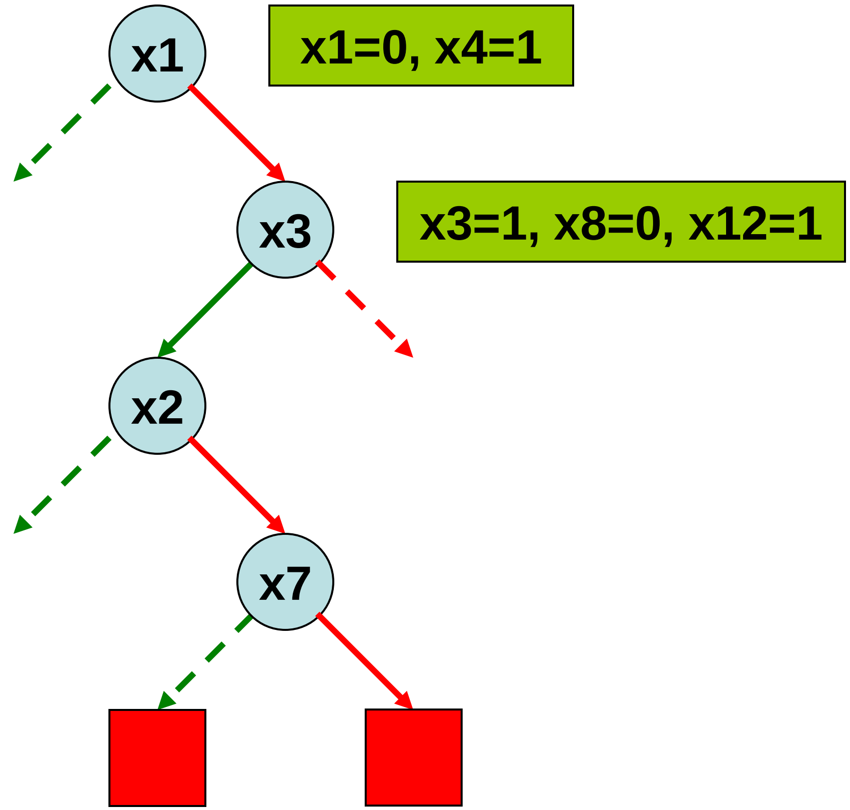


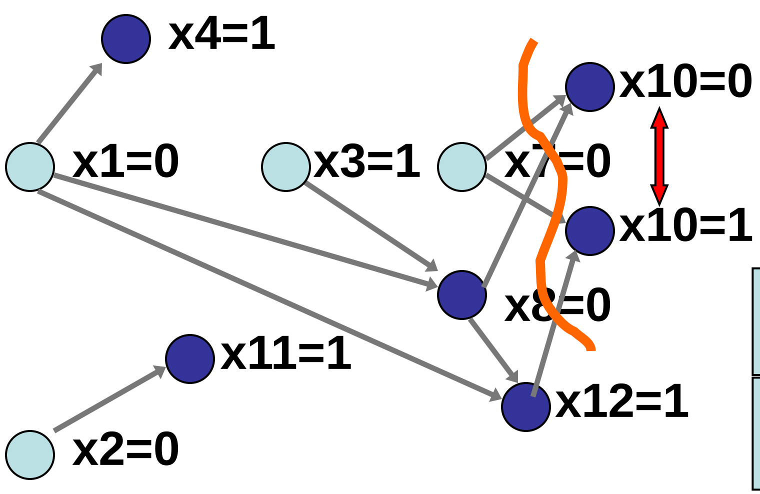
x3=1∧x7=1∧x8=0 → conflict
Add conflict clause: x3'+x7'+x8

Conflict Driven Learning

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'
x3' + x7' + x8

x7 + x8 + x12'



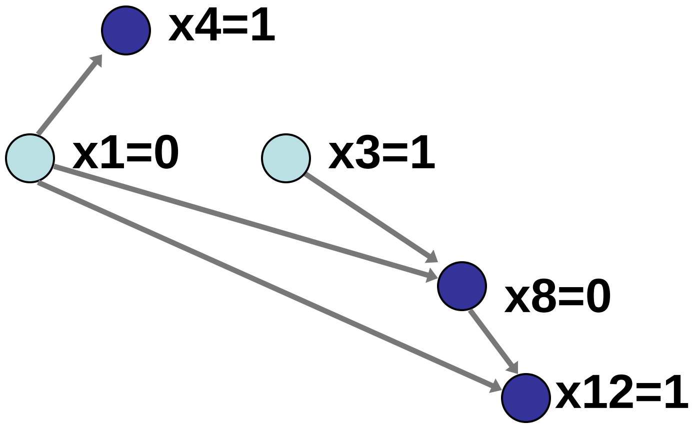


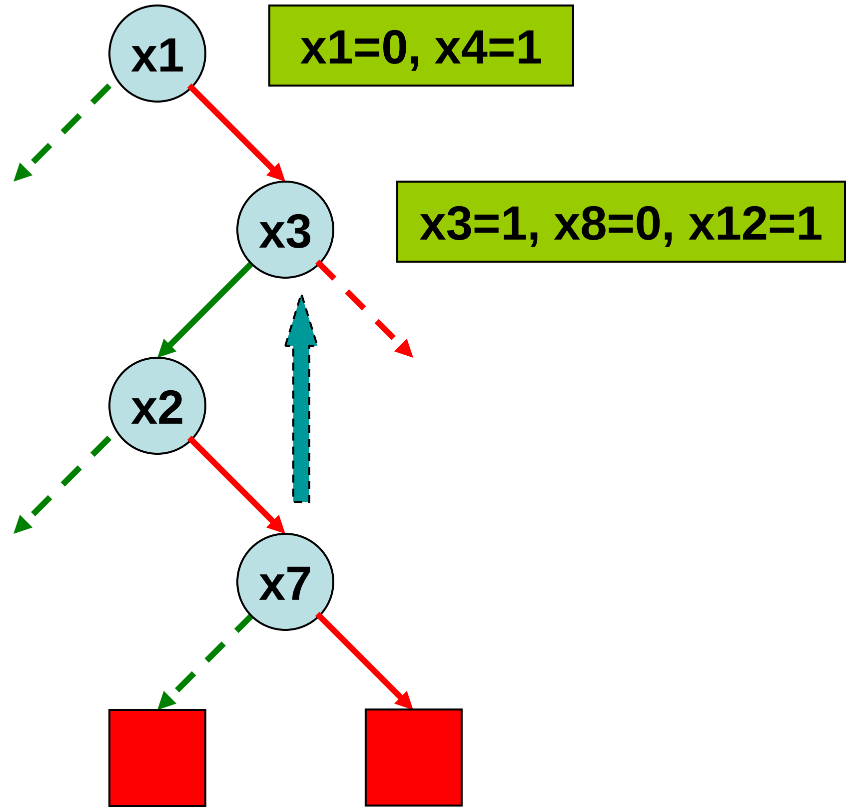
Backtrack and try x7 = 0

$x7=0 \wedge x8=0 \wedge x12=1 \rightarrow$ conflict

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'
x3' + x7' + x8
x7 + x8 + x12'





Backtrack to the decision level of x3=1, bypassing x2 since it is irrelevant

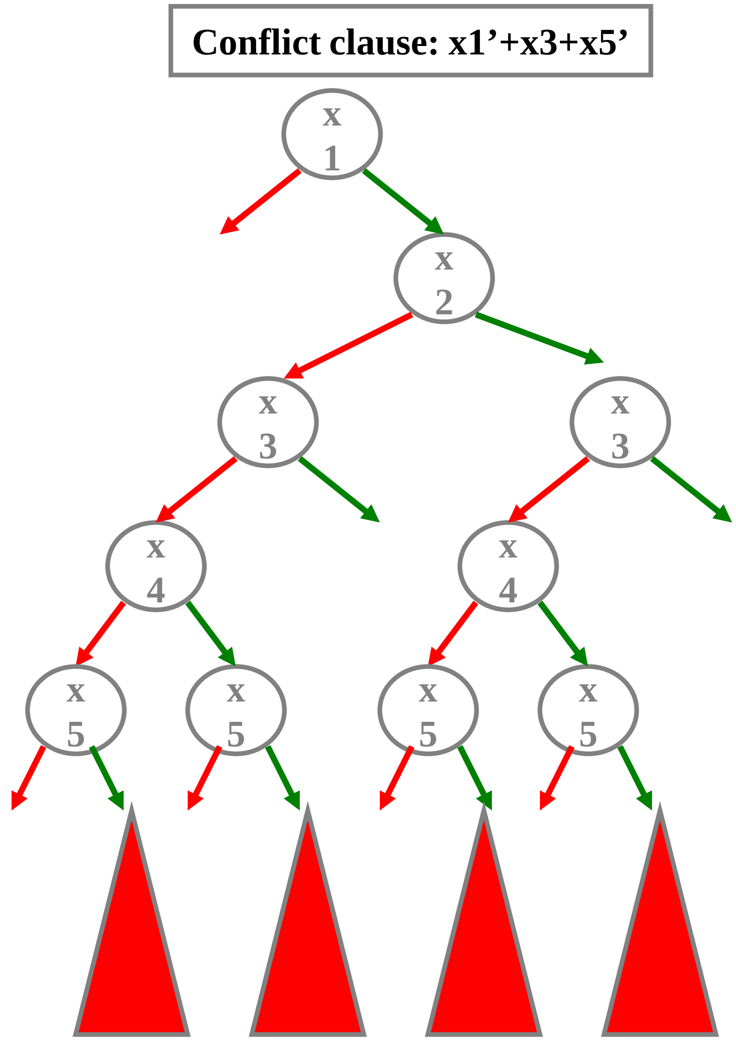
GRASP - General Principles

- **Conventional Approach**: Backtrack based on conflicts
- **New Approach**: Learning from conflicts (avoid repeating the same mistakes)

- **Conventional Approach**: Backtrack to the last decision
- **New Approach**: Backtracking based on analysis of the conflict (non-chronological)

Learning Conflict Clauses : What's the big deal?

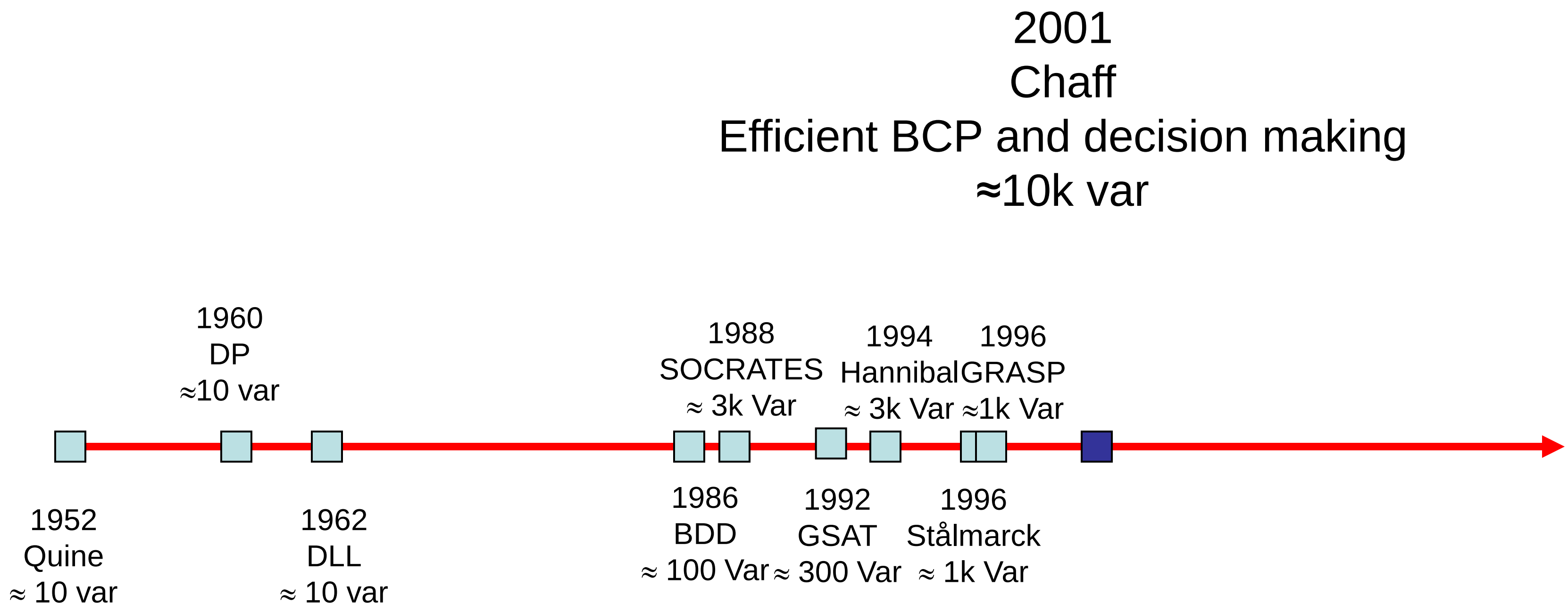
- Significantly prune the search space - learned clause helps repeat mistakes!
- Useful in generating future implications nad conflict clauses.
- Practical consideration – additional clauses require more memory
  - Limit the size of the clause
  - Limit the "lifetime" of a clause, will be removed after some time



SAT becomes practical!

- Conflict driven learning greatly increased the capacity of SAT solvers (several thousand variables) for structured problems
- Realistic applications became feasible
  - Typical EDA applications that can make use of SAT
    - ATPG
    - Circuit verification
    - FPGA routing
    - Covering (MIN-SAT)
    - …
- Research direction shifted towards more efficient implementations

The Timeline



M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik,"Chaff: Engineering an Efficient SAT Solver" *Proc. Design Automation Conference,* 2001.

Chaff Philosophy

- Make the core operations fast
  - Most time-consuming parts of a SAT solver
    - Boolean Constraint Propagation (BCP) and Decision Making
- Emphasis on coding efficiency
- Emphasis on optimizing data cache behavior
- As always, good search space pruning (i.e. conflict resolution and learning) is important

Motivation

	1dlx_c_mc_ex_bp_f
Num Variables	**776**
Num Clauses	**3725**
Num Literals	**10045**

	Z-Chaff	SATO	GRASP
# Decisions	3166	3771	1795
# Instructions	86.6M	630.4M	1415.9M
# L1/L2 accesses	24M / 1.7M	188M / 79M	416M / 153M
% L1/L2 misses	4.8% / 4.6%	36.8% / 9.7%	32.9% / 50.3%
# Seconds	0.22	4.41	11.78

Not sufficient to minimize decisions, need to consider implementation efficiency

Motivation (contd.)

- Need to think “differently” for large problem scales!

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

Efficient Boolean Constraint Propagation (BCP)

- Think of data structures used in a SAT solver
- **Formula** : List of clauses
- **Clause** : List of literals
- **Variable** : State + List of clauses it appears in (possibly separate lists for positive and negative phase)
- At any point during the search, how do you identify unit clauses?

Simple approach:

```
for each clause {
        count = 0;
        for each literal in clause {
            if (unassigned) count++;
            if (1) { not unit clause; }
        }
        if(count == 1) unit clause;
        else not unit clause;
}
```

Better approach (how would you do it?):

Efficient BCP : How Chaff does it

What "causes" an implication? When can it occur?

- All literals in a clause but one are False
  - (v1 + v2 + v3): implied cases: (0 + 0 + v3) or (0 + v2 + 0) or (v1 + 0 + 0)
- For an N-literal clause, this can only occur after N-1 of the literals are False
- So, 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: (v1 + v2 + v3 + v4 + v5)
  - (**v1=X + v2=X** + v3=? {i.e. X or 0 or 1} + v4=? + v5=?)

watch

BCP in Chaff (1/8)

- Invariants
  - Each clause has two watched literals.
  - If a clause can become newly implied via any sequence of assignments, then this sequence must include an assignment of one of the watched literals to F.
    - Example again: (v1 + v2 + v3 + v4 + v5)
    - (**v1=X** + **v2=X** + v3=? + v4=? + v5=?)

- BCP consists of identifying implied clauses (and the associated implications) while maintaining the "Invariants"

BCP in Chaff (2/8)

- Let’s illustrate this with an example:

```
v2 + v3 + v1 + v4 + v5
v1 + v2 + v3’
v1 + v2’
v1’+ v4
v1’
```

BCP in Chaff (2.1/8)

- Let's illustrate this with an example:

watched literals →

```
v2 + v3 + v1 + v4 + v5
v1 + v2 + v3'
v1 + v2'
v1'+ v4
v1'
```

← One literal clause breaks invariants: handled as a special case (ignored hereafter)

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

BCP in Chaff (3/8)

- We begin by processing the assignment v1 = F (which is implied by the size one clause)

State:(v1=F)
Pending:

v2 + v3 + v1 + v4 + v5

v1 + v2 + v3'

v1 + v2'

v1'+ v4

BCP in Chaff (3.1/8)

- We begin by processing the assignment v1 = F (which is implied by the size one clause)

```
State:(v1=F)
Pending:
```

v2 + v3 + v1 + v4 + v5

⇨ v1 + v2 + v3'

⇨ v1 + v2'

v1' + v4

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

BCP in Chaff (3.2/8)

- We begin by processing the assignment v1 = F (which is implied by the size one clause)

State:(v1=F)

Pending:

v2 + v3 + v1 + v4 + v5

v1 + v2 + v3’

v1 + v2’

v1’+ v4

- 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 in Chaff (3.3/8)

- We begin by processing the assignment v1 = F (which is implied by the size one clause)

State:(v1=F)	⇨	v2 + v3 + v1 + v4 + v5
Pending:		v1 + v2 + v3’
		v1 + v2’
		v1’+ v4

- 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 v1 is not watched).

BCP in Chaff (4/8)

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

```
v2 + v3 + v1 + v4 + v5
v1 + v2 + v3'
v1 + v2'
v1'+ v4
```

```
State:(v1=F)
Pending:
```

BCP in Chaff (4.1/8)

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

v2 + v3 + v1 + v4 + v5	→	v2 + v3 + v1 + v4 + v5
v1 + v2 + v3'		v1 + v2 + v3'
v1 + v2'		v1 + v2'
v1' + v4		v1' + v4
State:(v1=F) Pending:		State:(v1=F) Pending:

- 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 in Chaff (4.2/8)

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

Before:

v2 + v3 + v1 + v4 + v5

v1 + v2 + v3'

v1 + v2'

v1' + v4

State:(v1=F)

Pending:

After:

v2 + v3 + v1 + v4 + v5

v1 + v2 + v3'

v1 + v2'

v1' + v4

State:(v1=F)

Pending:(v2=F)

- 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 in Chaff (5/8)

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

```
v2 + v3 + v1 + v4 + v5          v2 + v3 + v1 + v4 + v5
v1 + v2 + v3'                   v1 + v2 + v3'
v1 + v2'                        v1 + v2'
v1'+ v4                         v1'+ v4

State:(v1=F, v2=F)              State:(v1=F, v2=F)
Pending:                        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 in Chaff (6/8)

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

v2 + v3 + v1 + v4 + v5

v1 + v2 + v3’

v1 + v2’

v1’+ v4

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

Pending:

→

v2 + v3 + v1 + v4 + v5

v1 + v2 + v3’

v1 + v2’

v1’+ v4

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 in Chaff (7/8)

- Next, we process v4. We do nothing at all.

v2 + v3 + v1 + v4 + v5	→	v2 + v3 + v1 + v4 + v5
v1 + v2 + v3'		v1 + v2 + v3'
v1 + v2'		v1 + v2'
v1' + v4		v1' + v4
State:(v1=F, v2=F, v3=F, v4=T)		State:(v1=F, v2=F, v3=F, v4=T)

- Since there are no pending assignments, and no conflict, BCP terminates and we make a decision. Only v5 is unassigned. Let's say we decide to assign v5=F and proceed.

BCP in Chaff (8/8)

- Next, we process v5=F. We examine the first clause.

v2 + v3 + v1 + v4 + v5

v1 + v2 + v3’

v1 + v2’

v1’+ v4

State:(v1=F, v2=F, v3=F, v4=T, v5=F)

v2 + v3 + v1 + v4 + v5

v1 + v2 + v3’

v1 + v2’

v1’+ v4

State:(v1=F, v2=F, v3=F, v4=T, v5=F)

- The first clause is implied. However, the implication is v4=T, which is a duplicate (since v4=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.

Summary: BCP in Chaff

- Maintain two “watched” literals for each clause
- During search, process the clause for BCP only if one of the watched literals is set to False
  - Two cases
    - Unit clause
    - Find a new literal to be the watched literal

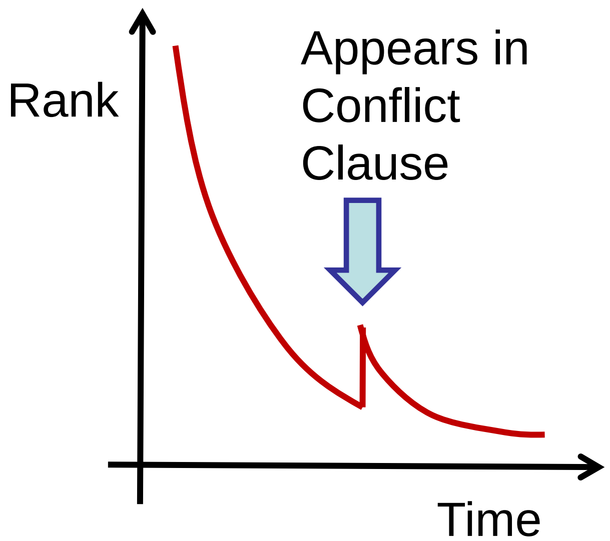
Decision Heuristics – Conventional Wisdom

- **D**ynamic **L**argest **I**ndividual **S**um (DLIS)
  - Simple and intuitive: At each decision, choose the assignment that satisfies the most unsatisfied clauses.
  - Considerable work is required to maintain the statistics necessary for this heuristic:
    - 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→1, update rankings.
    - Need to reverse the process for undoing an assignment.
  - The total effort required for this and similar decision heuristics is quite high.
- Look ahead algorithms are “smarter” but even more compute intensive

  C. Li, Anbulagan, “Look-ahead versus look-back for satisfiability problems” *Proc. Int. Conference on Principles and Practice of Constraint Programming*, 1997.

Chaff Decision Heuristic - VSIDS

- **V**ariable **S**tate **I**ndependent **D**ecaying **S**um
  - Each variable has two counters for each polarity
  - Only increment counts when new clauses are added to the CNF.
  - Periodically, divide all counts by a constant.
  - Variable and polarity with highest rank (counter value) chosen for branching.
    - Ties broken randomly
- Quasi-static:
  - Static : doesn't depend on variable state
  - VSIDS rank gradually changes as new clauses are added
    - Decay causes bias toward variables that appear in *recent* conflict clauses.
- Works reasonably in terms of # decisions
  - Much more efficient than state dependent heuristics



General Principles

- Need to consider implementation cost of a heuristic in addition to what it "saves".
- In the context of SAT, tradeoff between searching more and spending more time reasoning

Notable Recent Advances

- MiniSAT (http://minisat.se)
  - Continues philosophy of minimalistic design and focus on implementation efficiency
    - " An Extensible SAT-solver,“ Niklas Een, Niklas Sörensson, SAT 2003
    - " MiniSat — A SAT Solver with Conflict-Clause Minimization,“ Niklas Een, Niklas Sörensson, SAT 2005 (poster).
- Berkmin (http://eigold.tripod.com/BerkMin.html)
  - Improved heuristics for picking decision variables and clause database management
    - E.Goldberg, Y.Novikov, “BerkMin: a Fast and Robust SAT-Solver,” Design, Automation, and Test Europe, pp. 142-149, 2002.
- SatELite
  - Pre-processing formula to make solver more efficient
    - “Effective Preprocessing in SAT through Variable and Clause Elimination,” Niklas Een, Armin Biere, SAT 2005.

Summary

- Rich history of advances in SAT.
- Application drivers result in great progress.
- Need to account for computation cost of advanced heuristics
- Need to match algorithms with underlying computing platform architectures.
- Specific problem classes can benefit from specialized algorithms
- Much room to learn and improve!