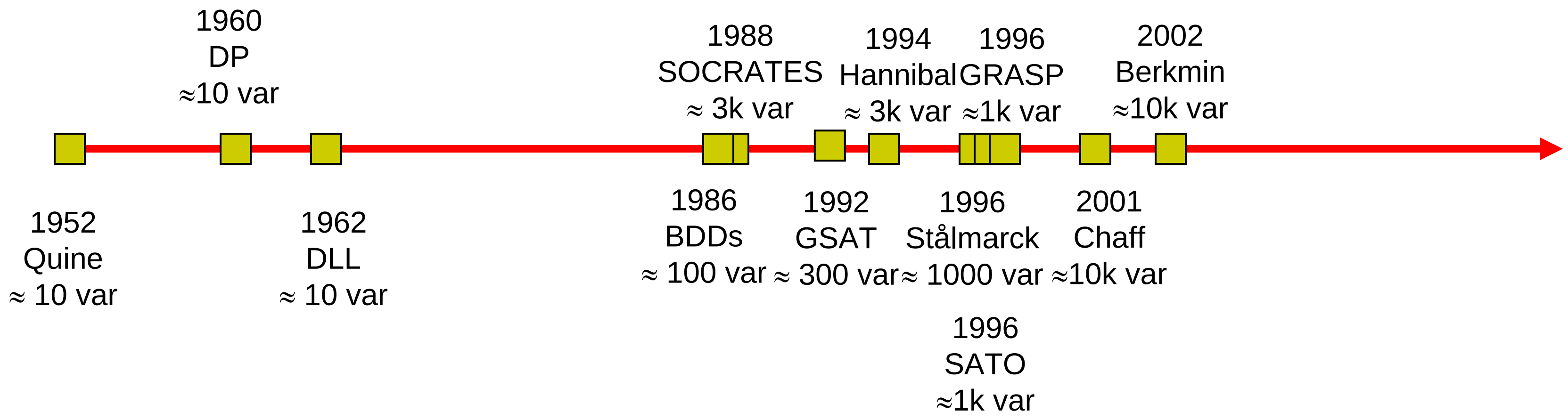
The Quest for Efficient Boolean Satisfiability Solvers

Sharad Malik
Princeton University

Acknowledgements

- Chaff authors:
  - Matthew Moskewicz (now at UC Berkeley)
  - Conor Madigan (now at MIT)
- Princeton University SAT group:
  - Daijue Tang
  - Yinlei Yu
  - Yogesh Mahajan
  - Zhaohui Fu
  - Lintao Zhang (now at Microsoft Research)

The Timeline

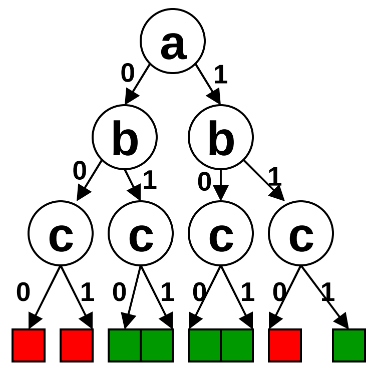


SAT in a Nutshell

- Given a Boolean formula (propositional logic formula), find a variable assignment such that the formula evaluates to 1, or prove that no such assignment exists.

F = (a + b)(a’ + b’ + c)

- For *n* variables, there are 2^n possible truth assignments to be checked.



- First established NP-Complete problem.

  S. A. Cook, The complexity of theorem proving procedures, *Proceedings, Third Annual ACM Symp. on the Theory of Computing*,1971, 151-158

Problem Representation

- Conjunctive Normal Form
  - F = (a + b)(a' + b' + c)

clause

literal

  - Simple representation (more efficient data structures)
- Logic circuit representation
  - Circuits have structural and direction information
- Circuit – CNF conversion is straightforward

d ≡ (a + b)

(a + b + d')

(a' + d)

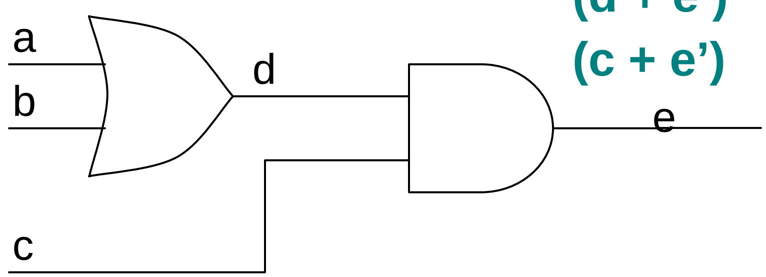
(b' + d)

e ≡ (c · d)

(c' + d' + e)

(d + e')

(c + e')



Why Bother?

- Core computational engine for major applications
  - EDA
    - Testing and Verification
    - Logic synthesis
    - FPGA routing
    - Path delay analysis
    - And more…
  - AI
    - Knowledge base deduction
    - Automatic theorem proving

The Timeline

1869: William Stanley Jevons: Logic Machine
[Gent & Walsh, SAT2000]

Pure Logic and other Minor Works –
Available at amazon.com!

The Timeline

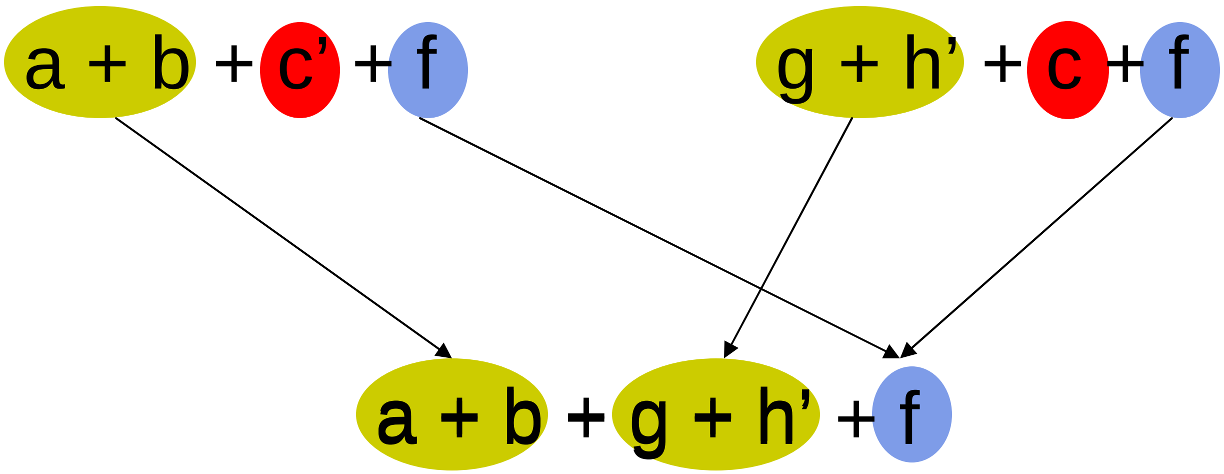
1960: Davis Putnam

Resolution Based

≈10 variables

Resolution

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



Davis Putnam Algorithm

M .Davis, H. Putnam, “A computing procedure for quantification theory", *J. of ACM*, Vol. 7, pp. 201-214, 1960 (360 citations in citeseer)

- 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)

∃b F = (a + c + e)(c’ + e + f)

∃bc F = (a + e + f)

∃bcaef F = 1

SAT

F = (a + b) (a + b’) (a’ + c) (a’ + c’)

∃b F = (a) (a’ + c) (a’ + c’)

∃ba F = (c) (c’)

∃bac F = ( )

UNSAT

Potential memory explosion problem!

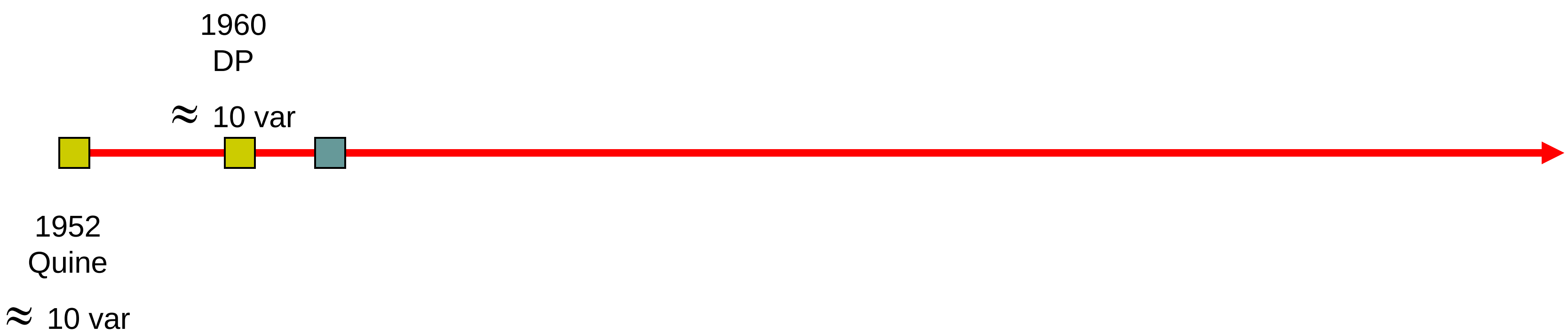
The Timeline

1952
Quine
Iterated Consensus
≈10 var

1960
DP
≈10 var

The Timeline

1962
Davis Logemann Loveland
Depth First Search
≈ 10 var



DLL Algorithm

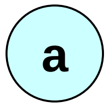
- Davis, Logemann and Loveland

  M. Davis, G. Logemann and D. Loveland, “A Machine Program for Theorem-Proving", *Communications of ACM*, Vol. 5, No. 7, pp. 394-397, 1962 (272 citations)
- Also known as DPLL for historical reasons
- Basic framework for many modern SAT solvers

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)

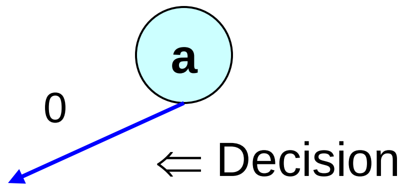
Basic DLL Procedure - DFS



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

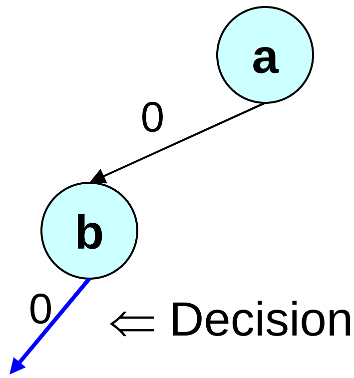
Basic DLL Procedure - DFS

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



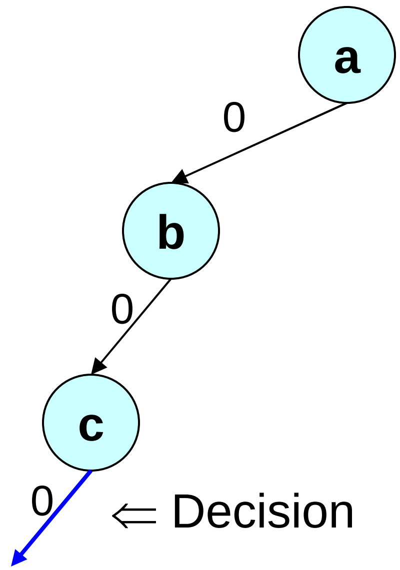
Basic DLL Procedure - DFS

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



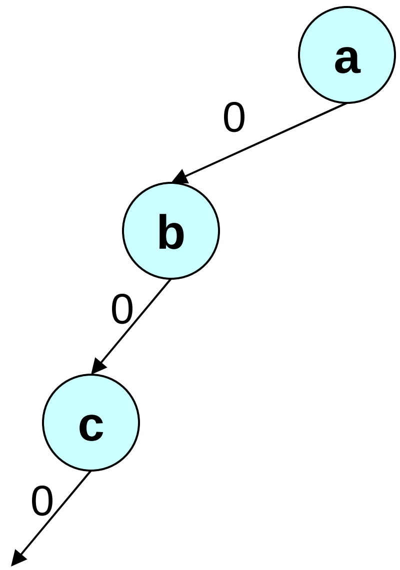
Basic DLL Procedure - DFS

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

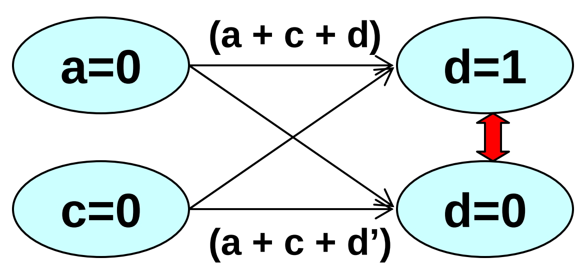


Basic DLL Procedure - DFS

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



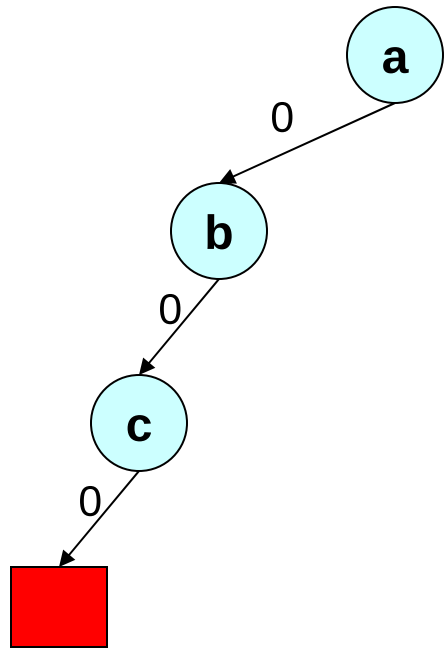
Implication Graph



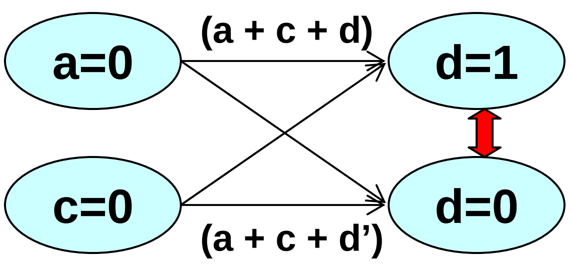
Conflict!

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)



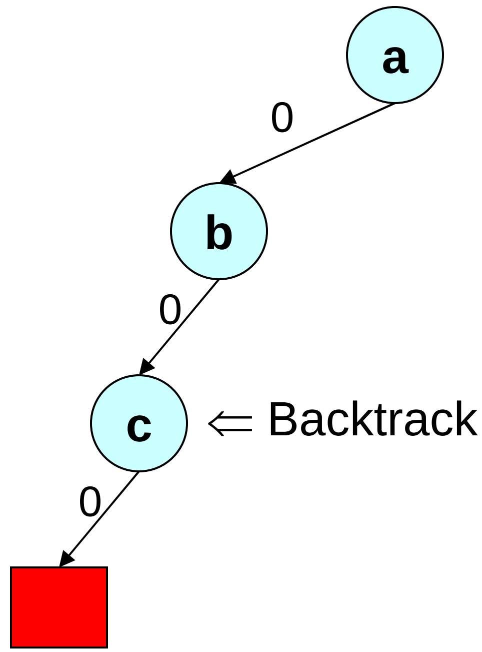
Implication Graph



Conflict!

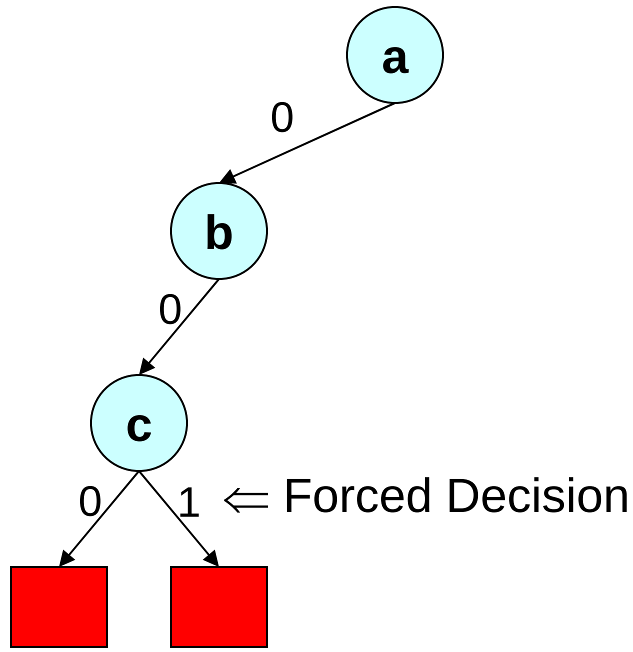
Basic DLL Procedure - DFS

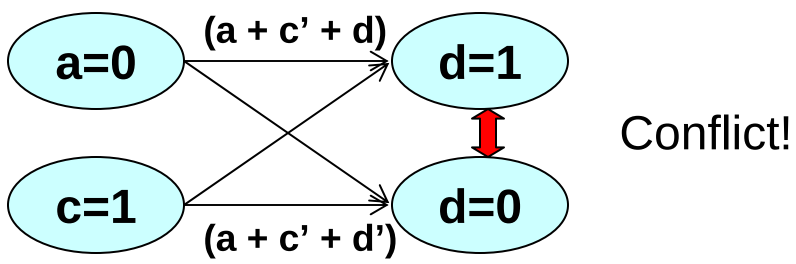
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

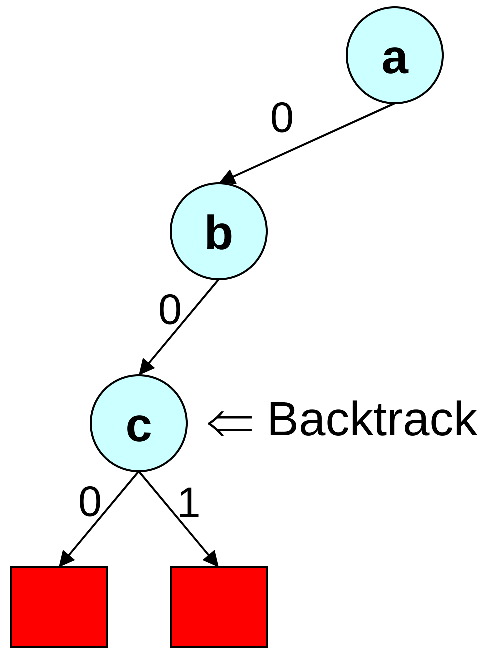
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)





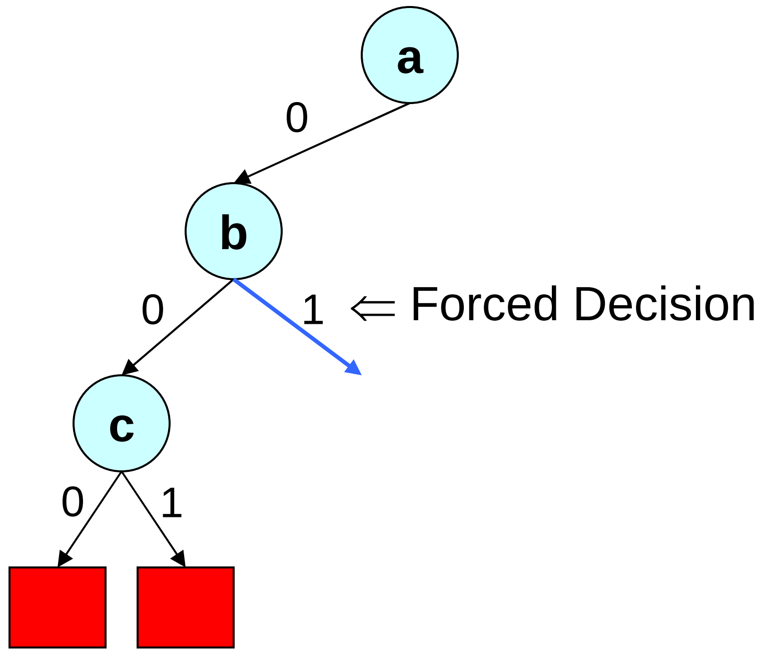
Basic DLL Procedure - DFS

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



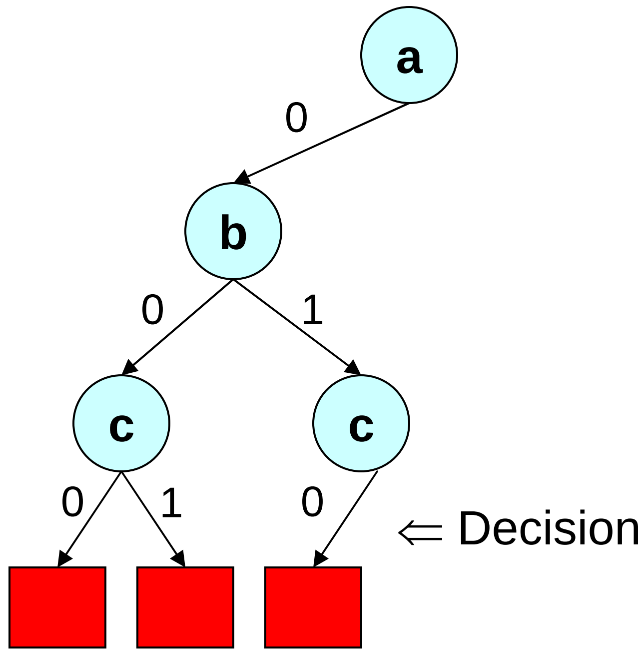
Basic DLL Procedure - DFS

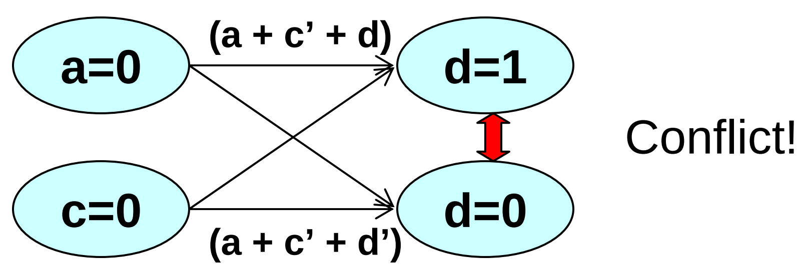
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

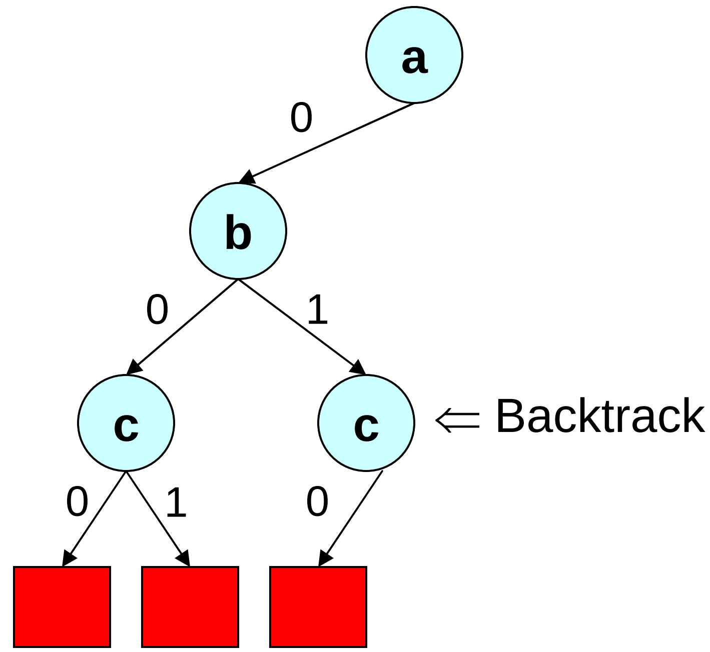
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)





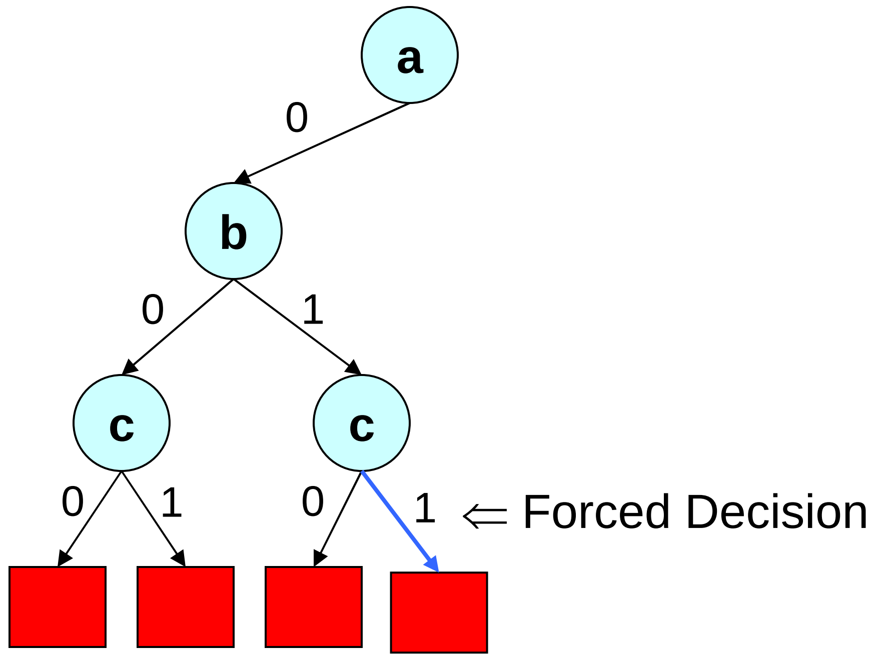
Basic DLL Procedure - DFS

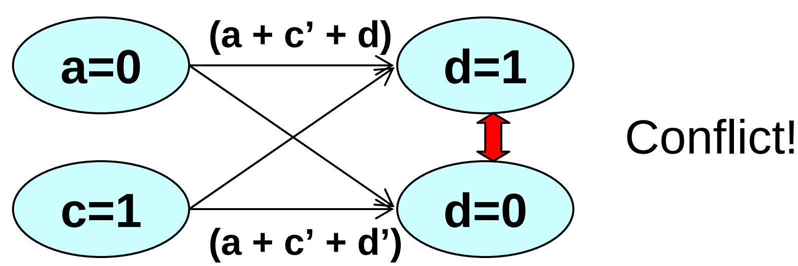
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

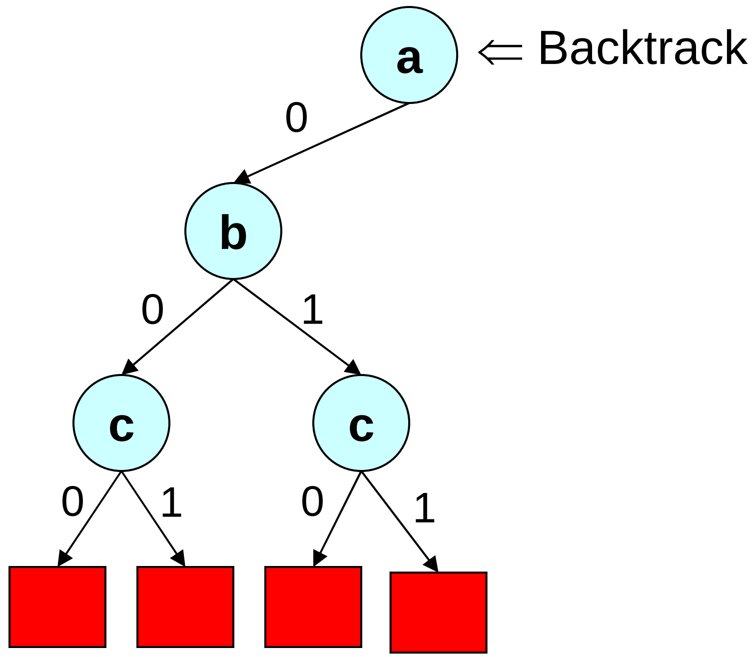
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)





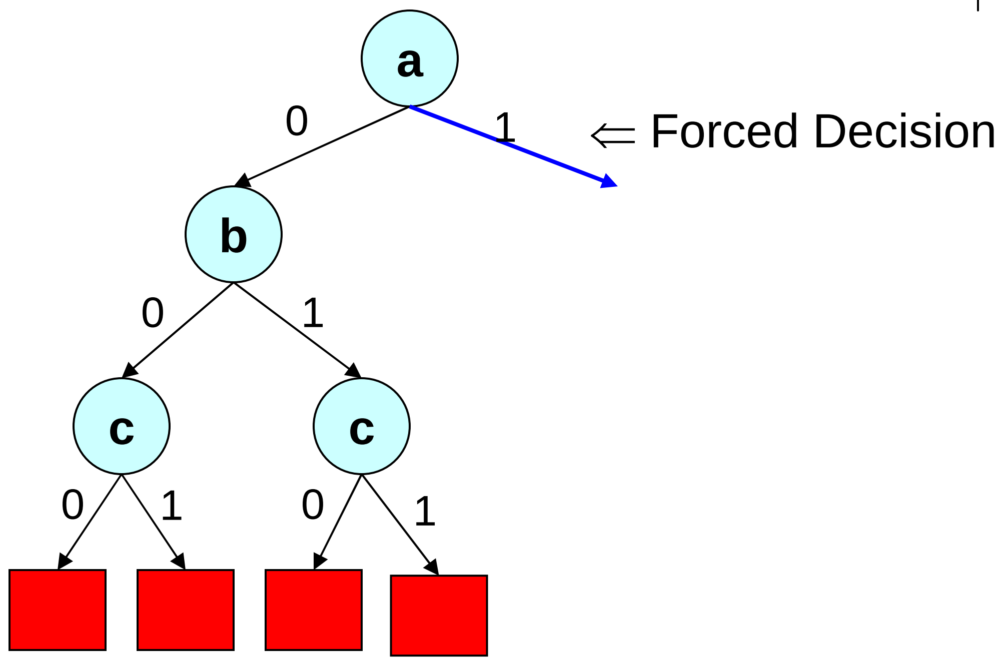
Basic DLL Procedure - DFS

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



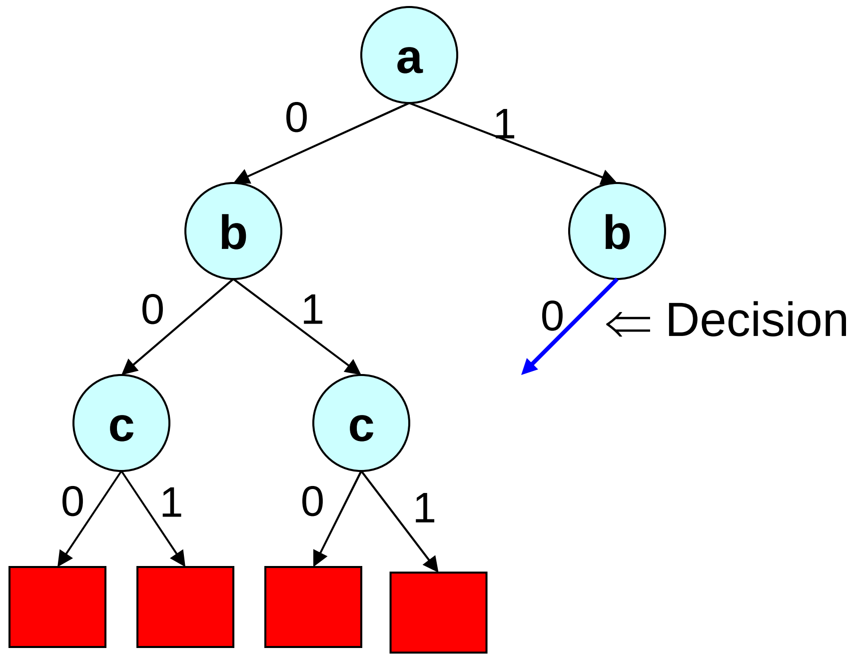
Basic DLL Procedure - DFS

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



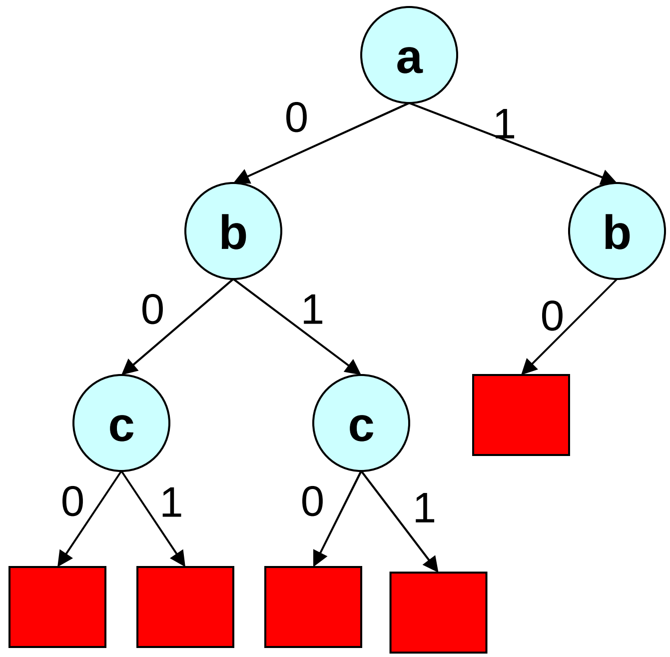
Basic DLL Procedure - DFS

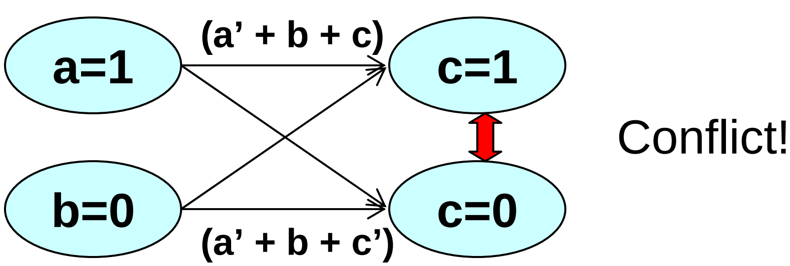
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



Basic DLL Procedure - DFS

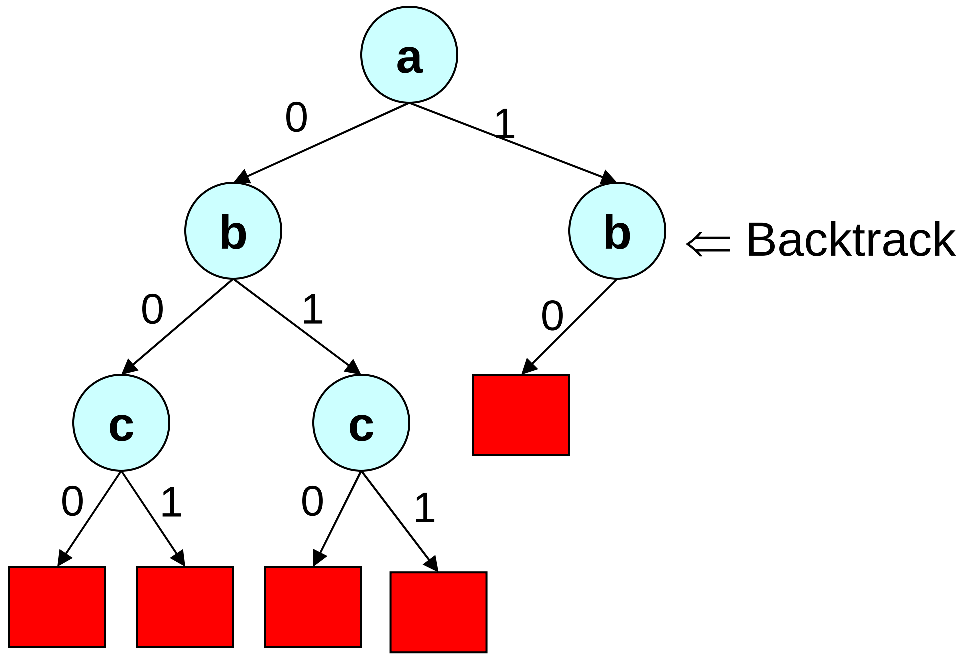
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)





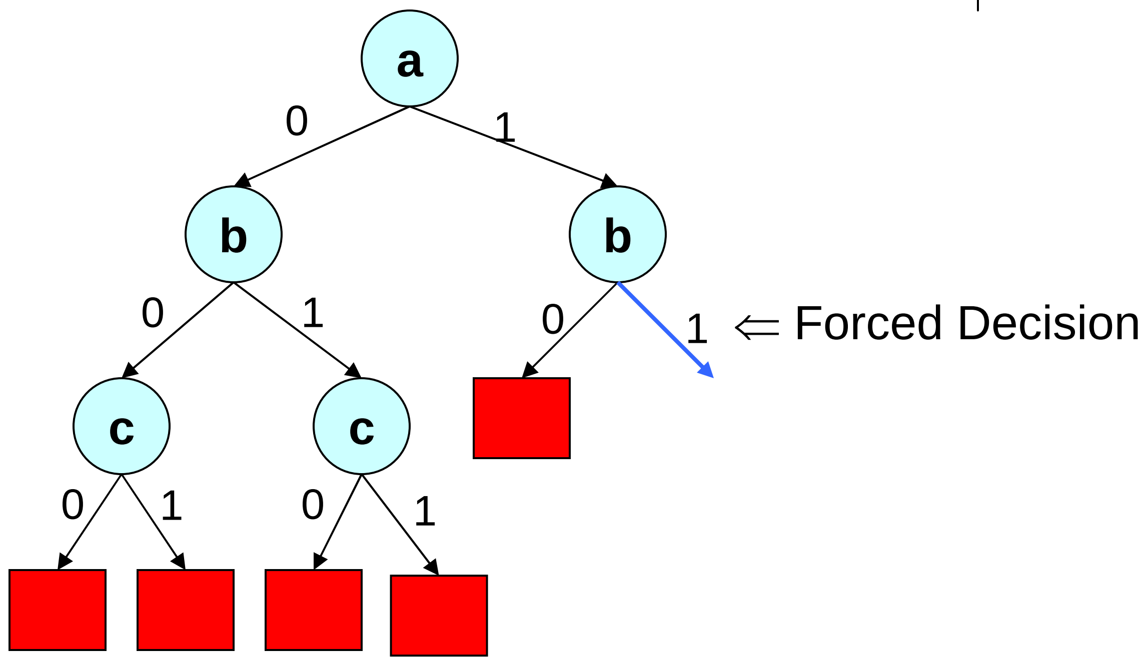
Basic DLL Procedure - DFS

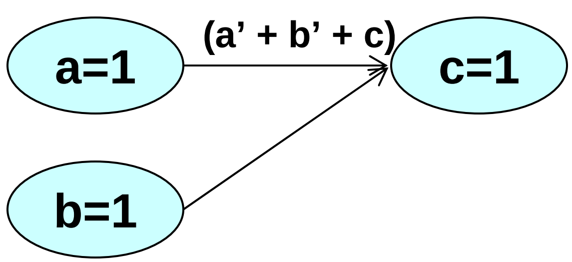
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

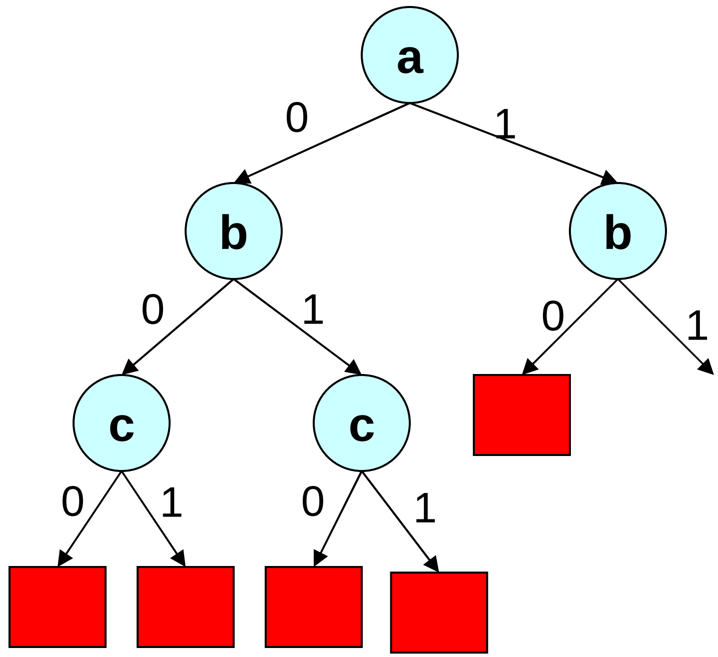
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

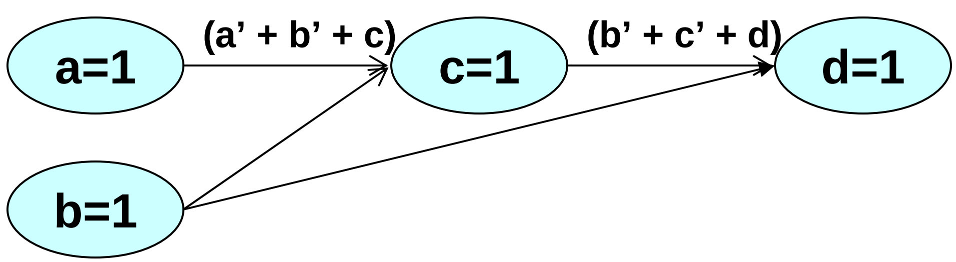




Basic DLL Procedure - DFS

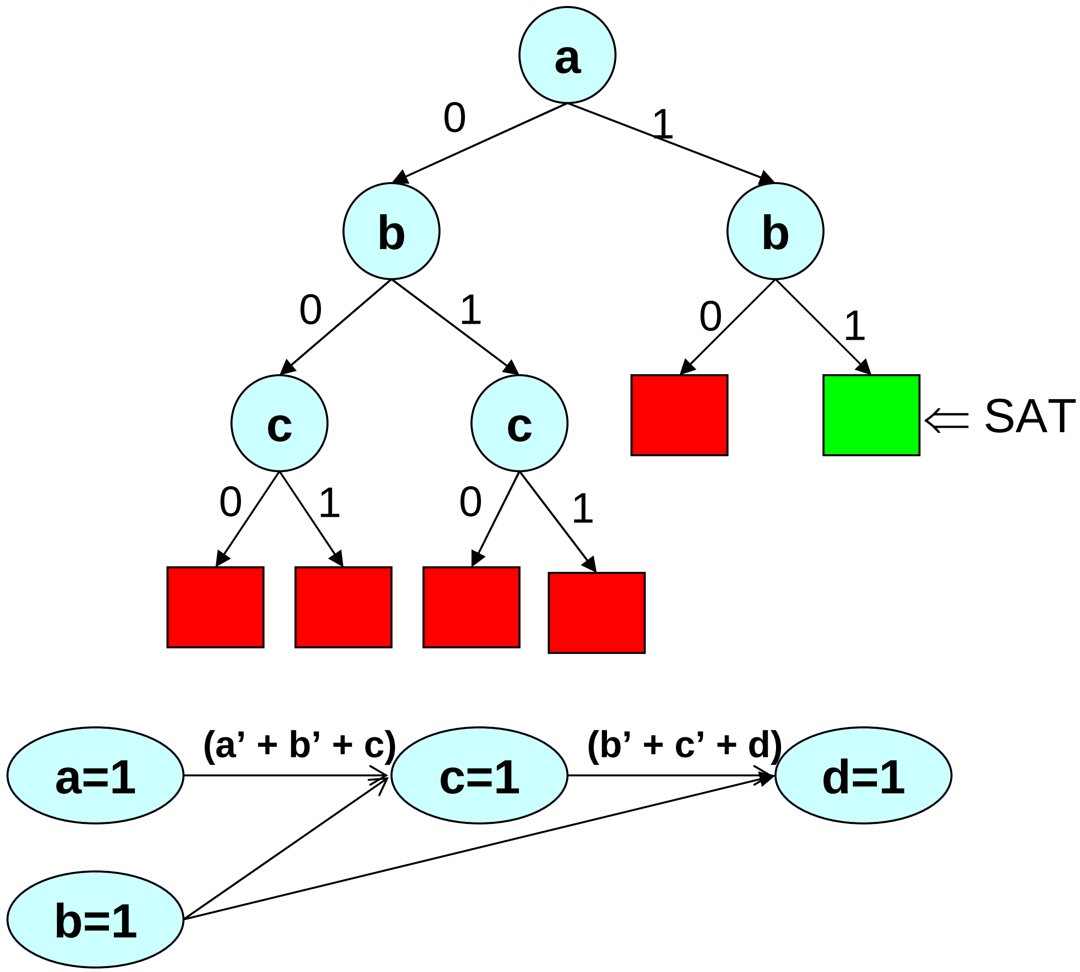
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)





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)



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 is unassigned

Satisfied Literal

Unsatisfied Literal

Unassigned Literal

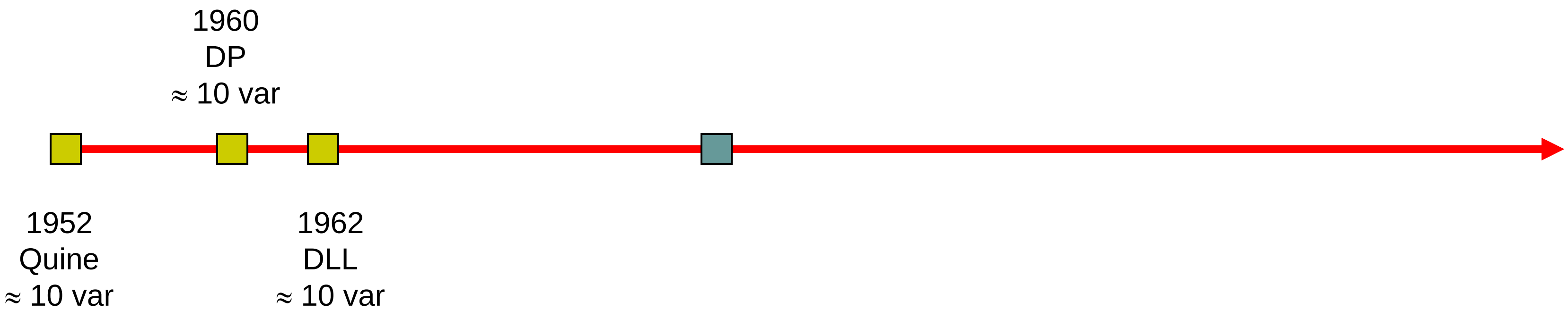
  - The unassigned literal is implied because of the unit clause.
- Boolean Constraint Propagation (BCP)
  - Iteratively apply the unit clause rule until there is no unit clause available.
  - a.k.a. Unit Propagation
- Workhorse of DLL based algorithms.

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)

The Timeline

1986
Binary Decision Diagrams (BDDs)
≈100 var

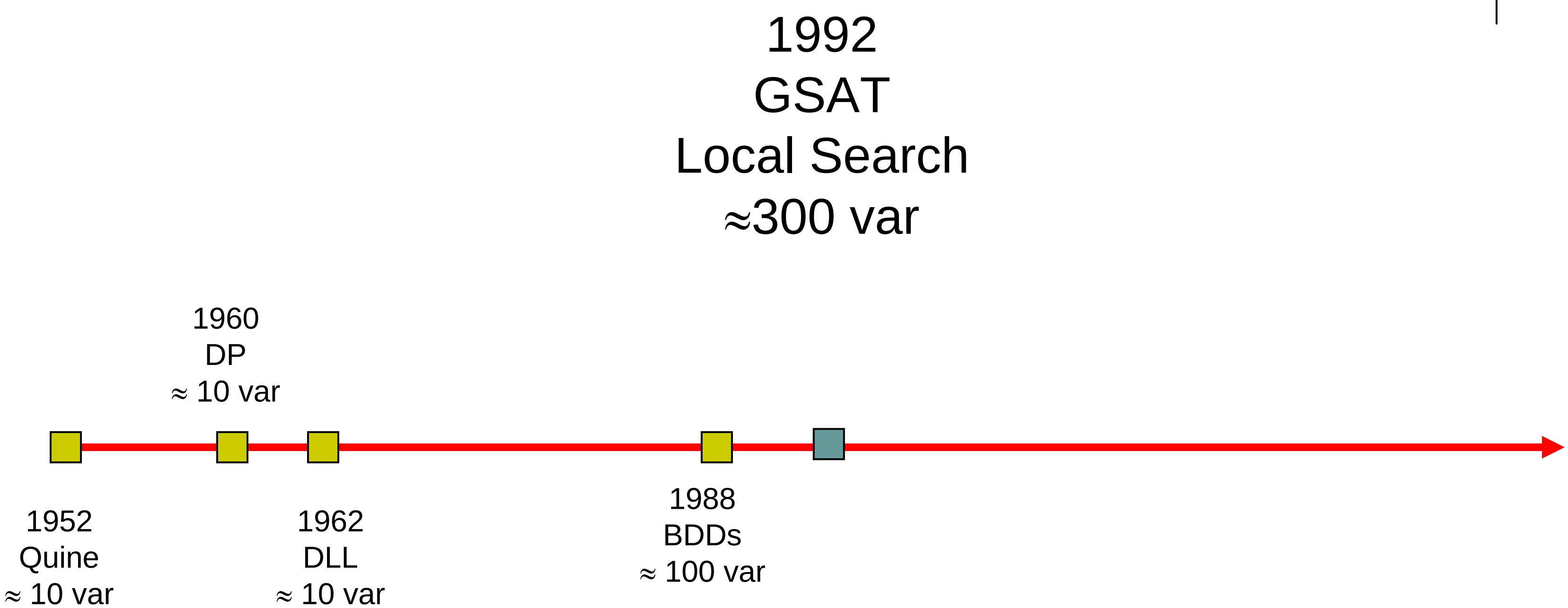


Using BDDs to Solve SAT

R. Bryant. “Graph-based algorithms for Boolean function manipulation”. *IEEE Trans. on Computers*, C-35, 8:677-691, 1986. (1308 citations)

- 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.

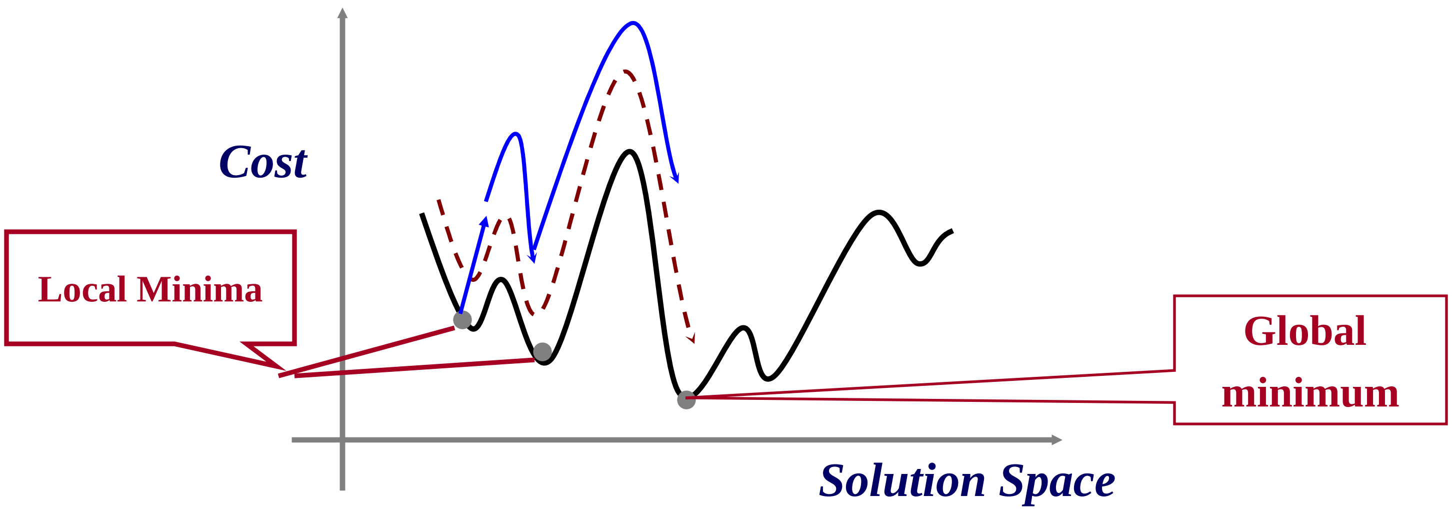
The Timeline



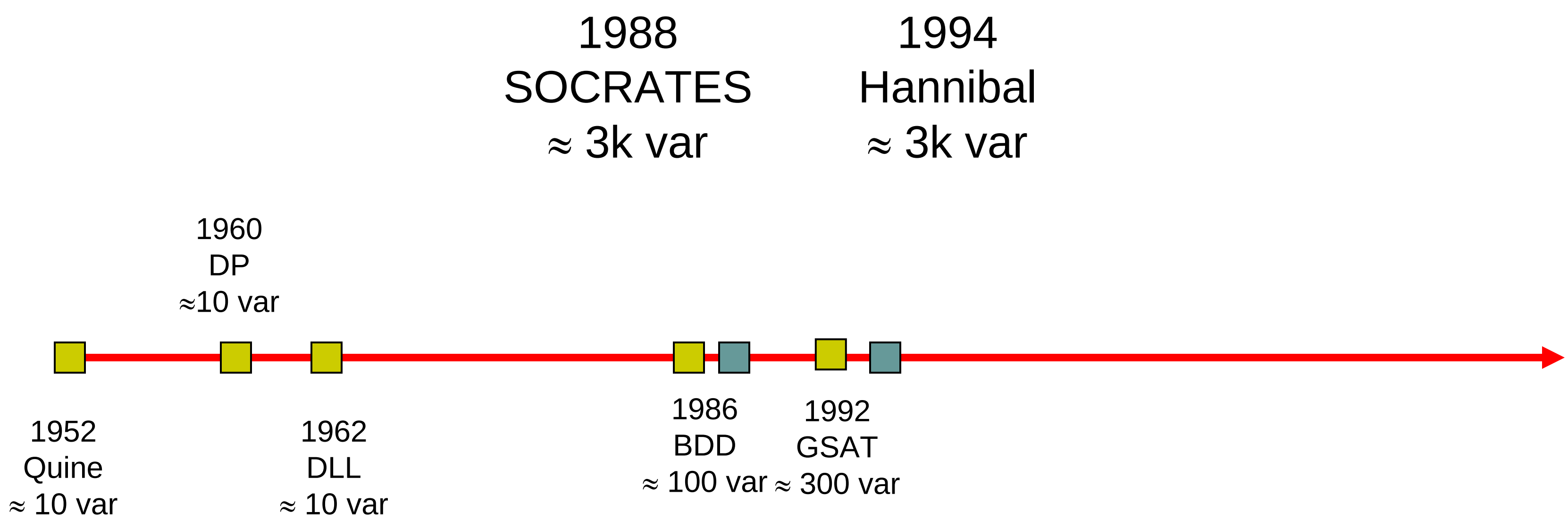
Local Search (GSAT, WSAT)

B. Selman, H. Levesque, and D. Mitchell. "A new method for solving hard satisfiability problems". *Proc. AAAI*, 1992. (373 citations)

- Hill climbing algorithm for local search
  - State: complete variable assignment
  - Cost: number of unsatisfied clauses
  - Move: flip one variable assignment
- Probabilistically accept moves that worsen the cost function to enable exits from local minima
- Incomplete SAT solvers
  - Geared towards satisfiable instances, cannot prove unsatisfiability



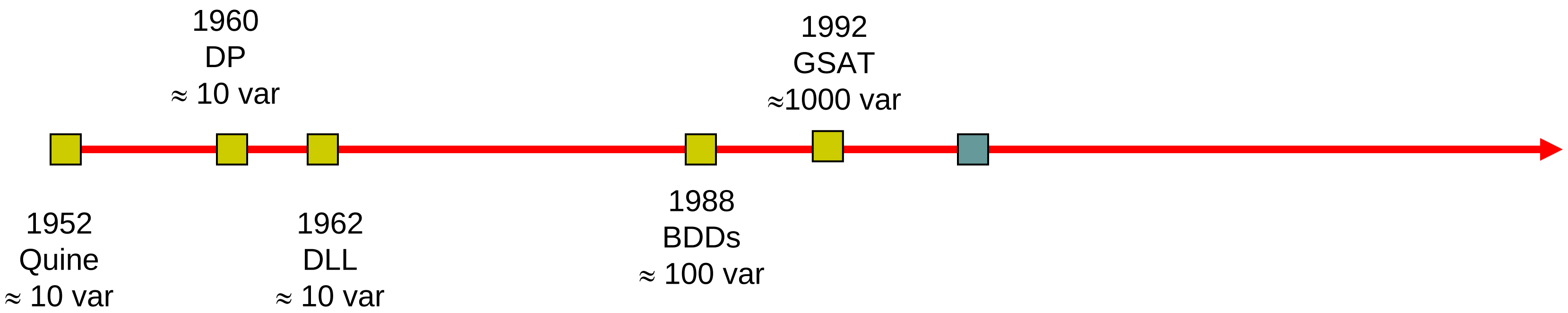
The Timeline



EDA Drivers (ATPG, Equivalence Checking)
start the push for practically useable algorithms!
Deemphasize random/synthetic benchmarks.

The Timeline

1996
Stålmarck's Algorithm
≈1000 var



Stålmarck's Algorithm

M. Sheeran and G. Stålmarck "A tutorial on Stålmarck's proof procedure", *Proc. FMCAD*, 1998 (10 citations)

- Algorithm:
  - Using triplets to represent formula
    - Closer to a circuit representation
  - Branch on variable relationships besides on variables
    - Ability to add new variables on the fly
  - Breadth first search over all possible trees in increasing depth

Stålmarck's algorithm

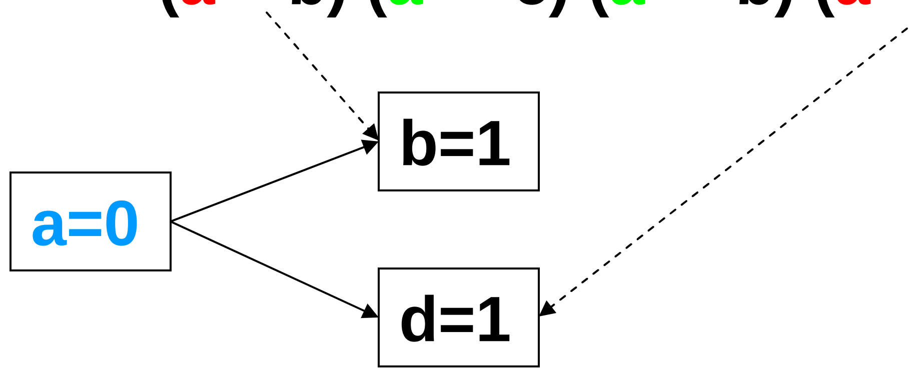
- Try both sides of a branch to find forced decisions (relationships between variables)

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

Stålmarck's algorithm

- Try both sides of a branch to find forced decisions

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

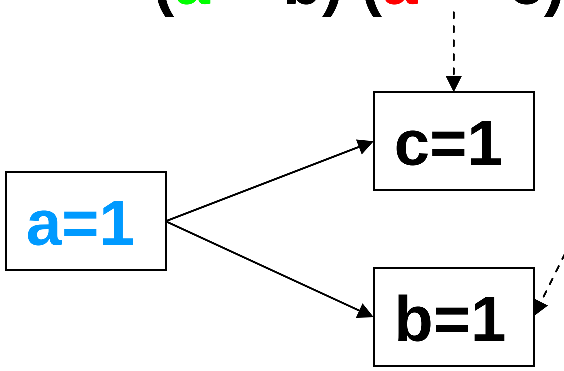


$a=0 \Rightarrow b=1, d=1$

Stålmarck's algorithm

- Try both side of a branch to find forced decisions

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



a=0 ⇒b=1,d=1

a=1 ⇒b=1,c=1

Stålmarck's algorithm

- Try both sides of a branch to find forced decisions

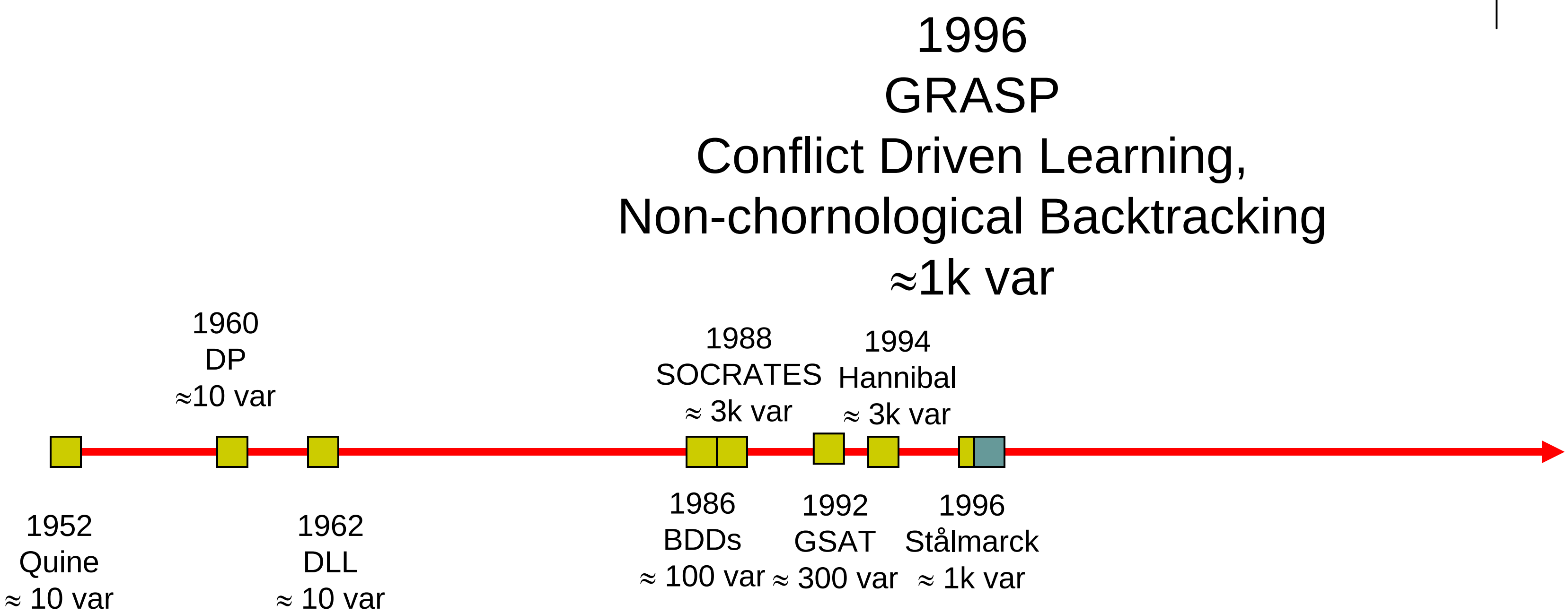
(a + b) (a' + c) (a' + b) (a + d)

a=0 $\Rightarrow$ b=1,d=1
a=1 $\Rightarrow$ b=1,c=1

$\Rightarrow$ **b=1**

- Repeat for all variables
- Repeat for all pairs, triples,… till either SAT or UNSAT is proved

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. (58 citations)

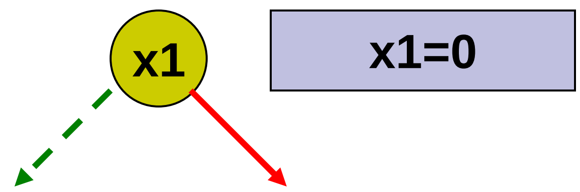
  J. P. Marques-Silva and Karem A. Sakallah, “GRASP: A Search Algorithm for Propositional Satisfiability”, *IEEE Trans. Computers*, C-48, 5:506-521, 1999. (19 citations)
- Incorporates conflict driven learning and non-chronological backtracking
- Practical SAT instances can be solved in reasonable time
- 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)

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’

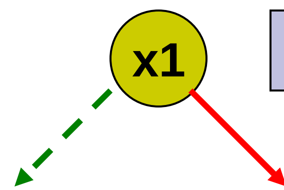
Conflict Driven Learning and Non-chronological Backtracking



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

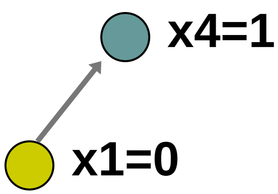
● x1=0

Conflict Driven Learning and Non-chronological Backtracking



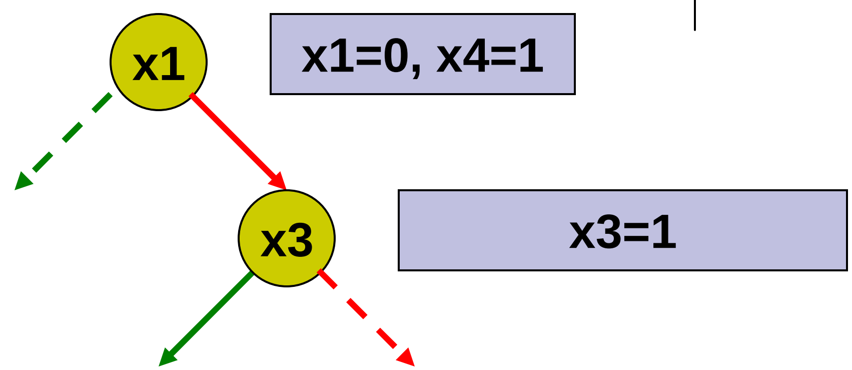
x1=0, x4=1

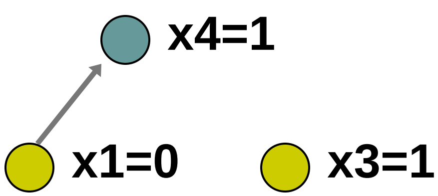
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’



Conflict Driven Learning and Non-chronological Backtracking

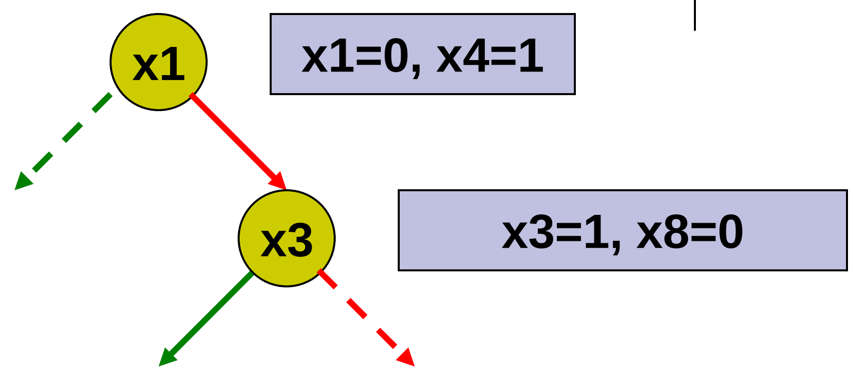
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

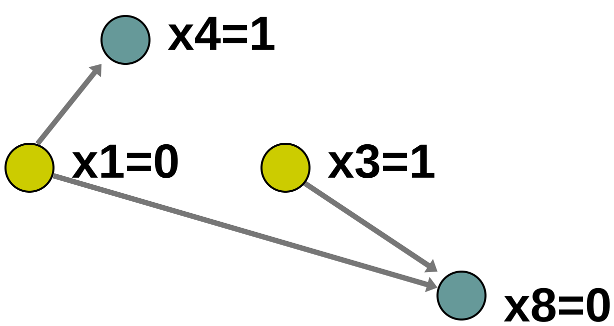




Conflict Driven Learning and Non-chronological Backtracking

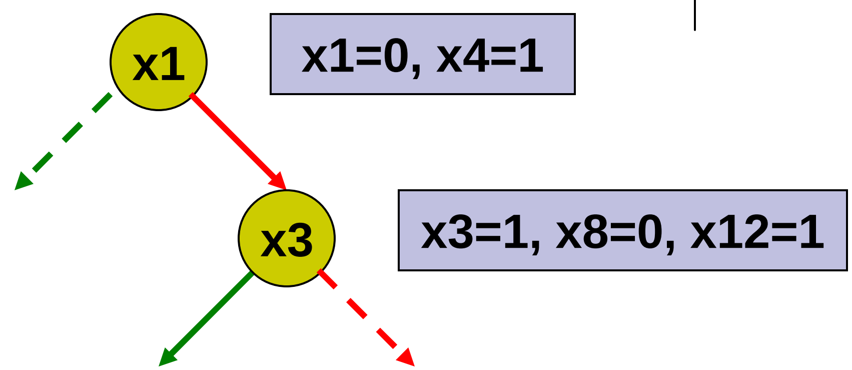
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

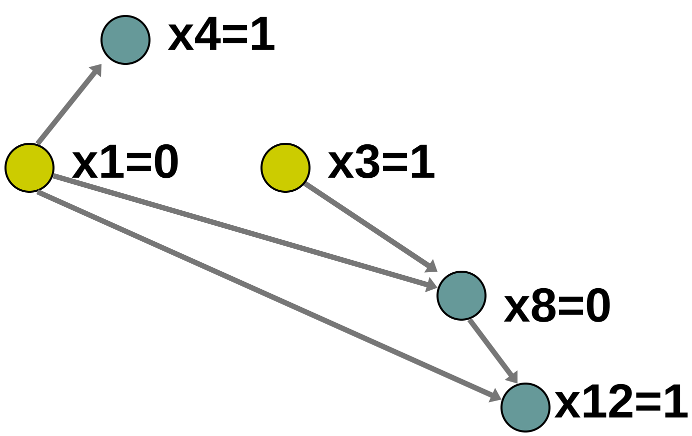




Conflict Driven Learning and Non-chronological Backtracking

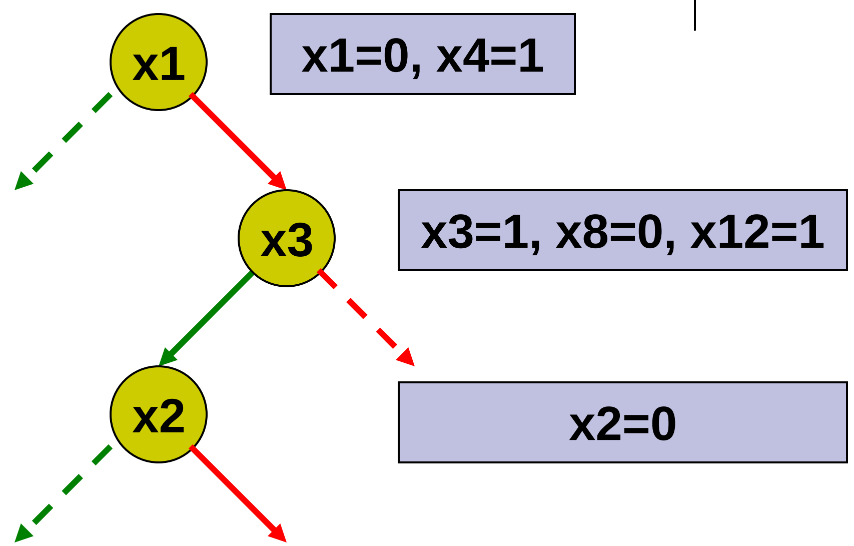
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

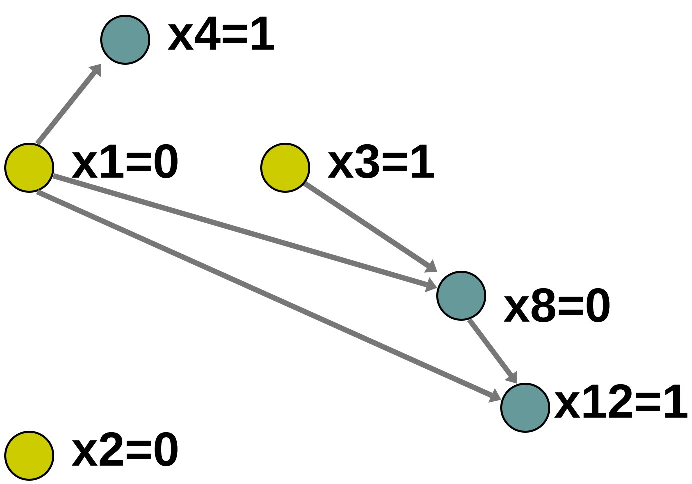




Conflict Driven Learning and Non-chronological Backtracking

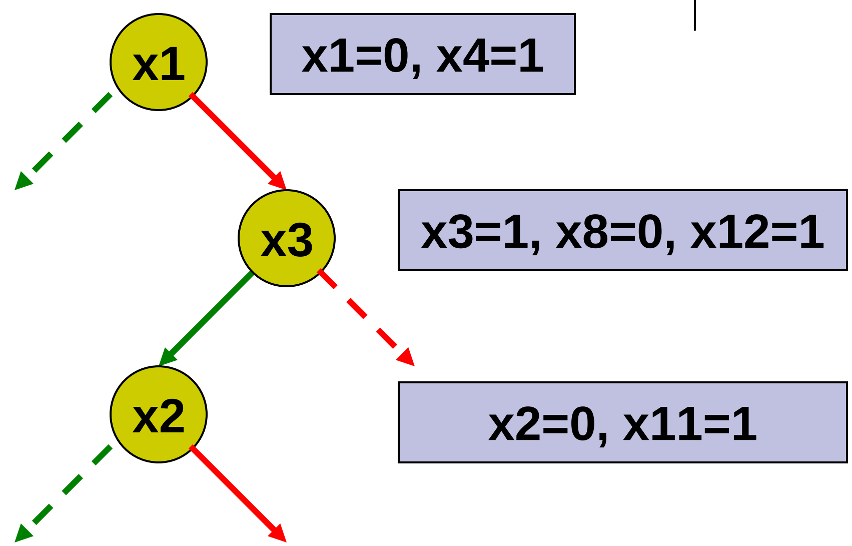
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

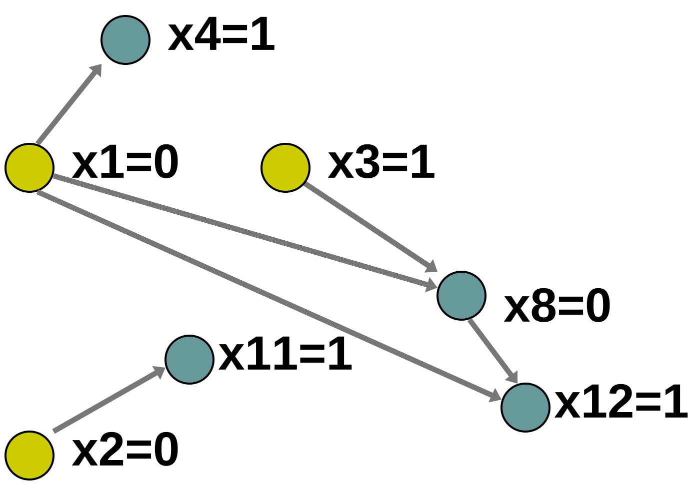




Conflict Driven Learning and Non-chronological Backtracking

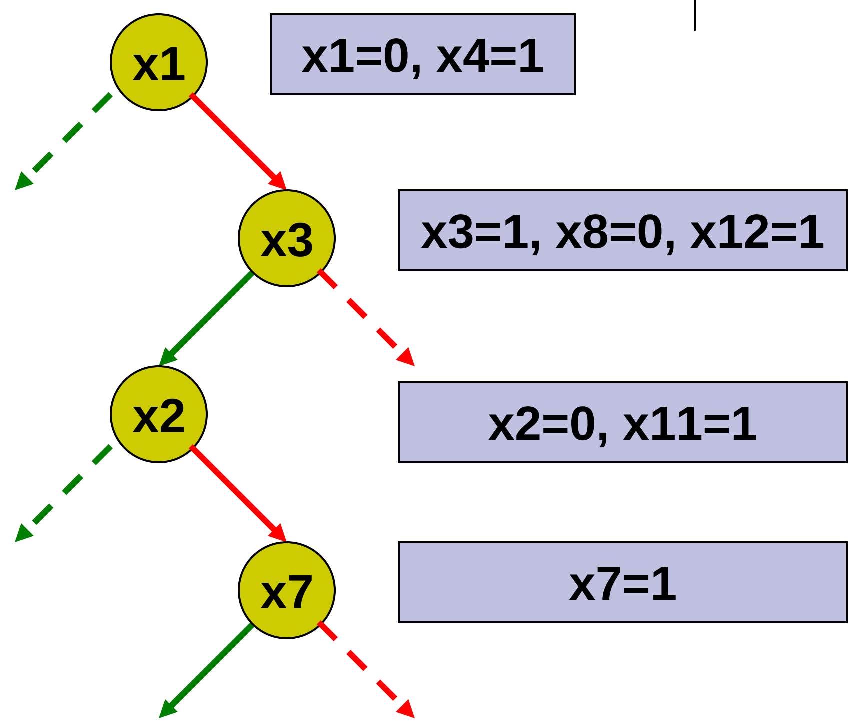
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

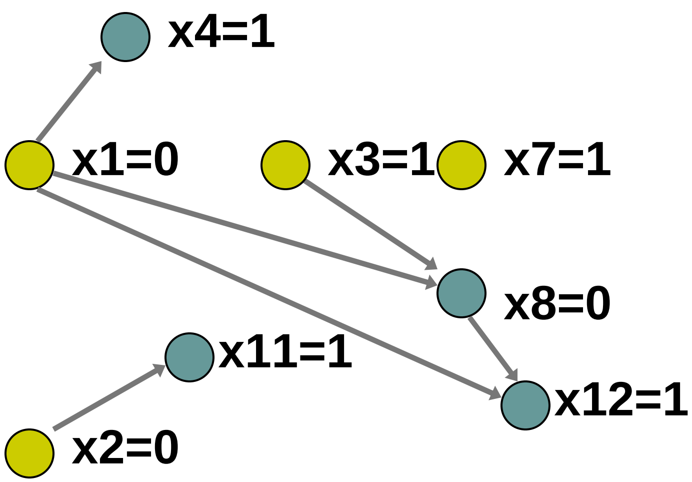




Conflict Driven Learning and Non-chronological Backtracking

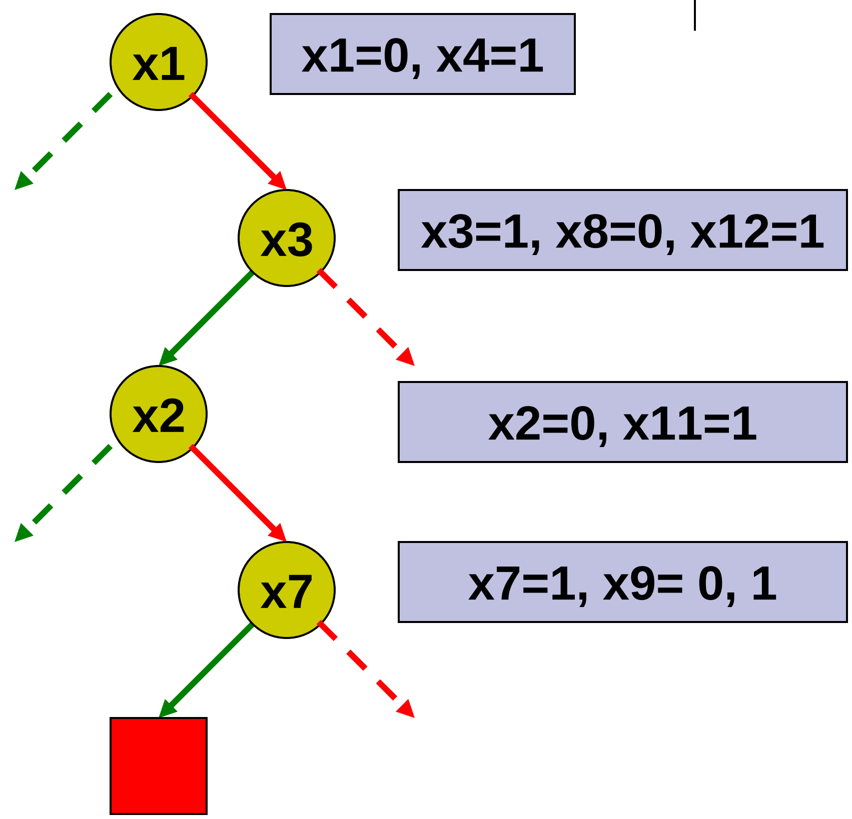
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

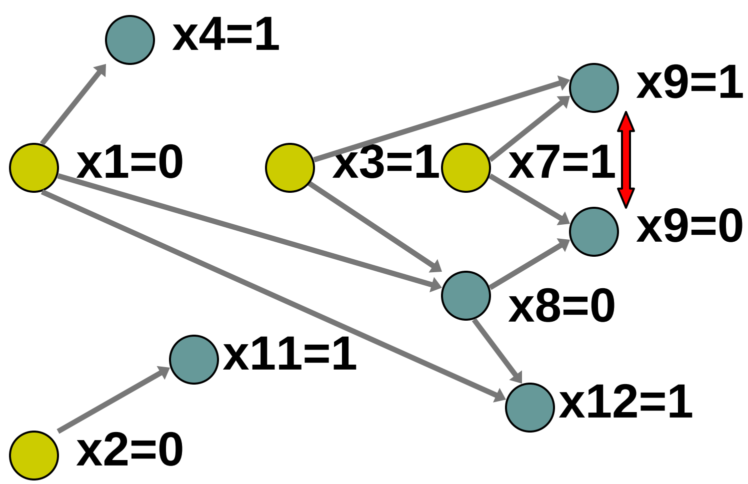




Conflict Driven Learning and Non-chronological Backtracking

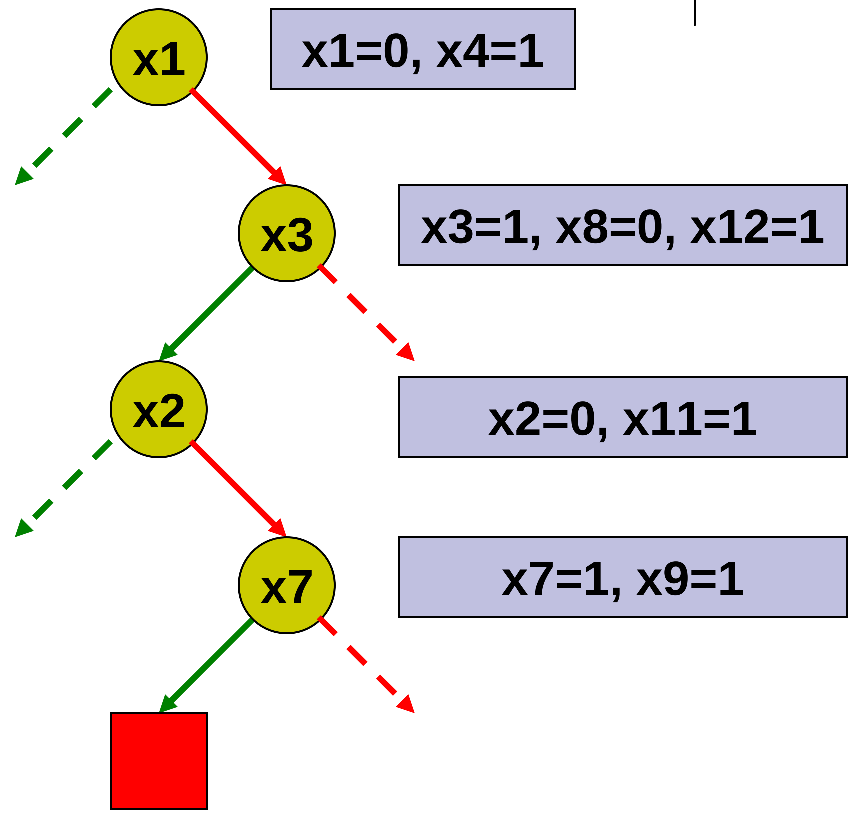
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’



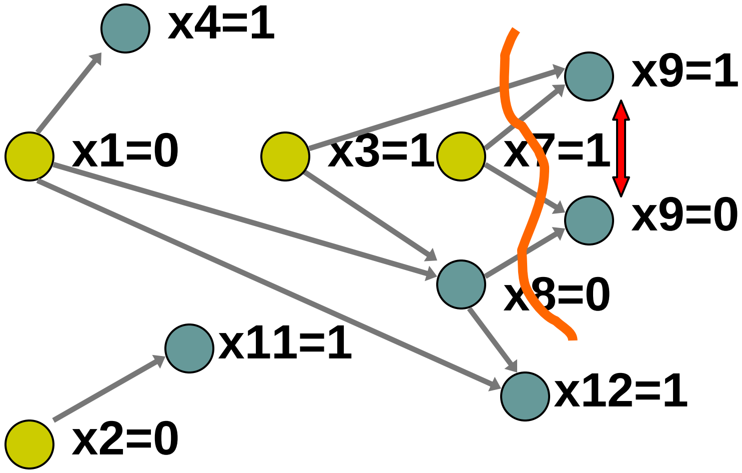


Conflict Driven Learning and Non-chronological Backtracking

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

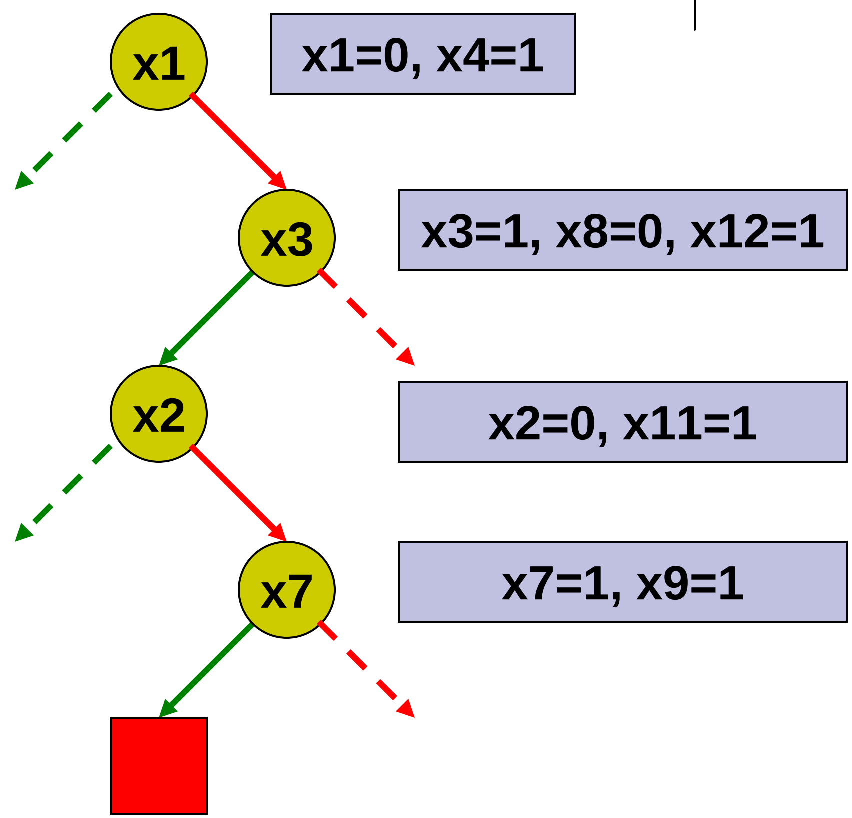


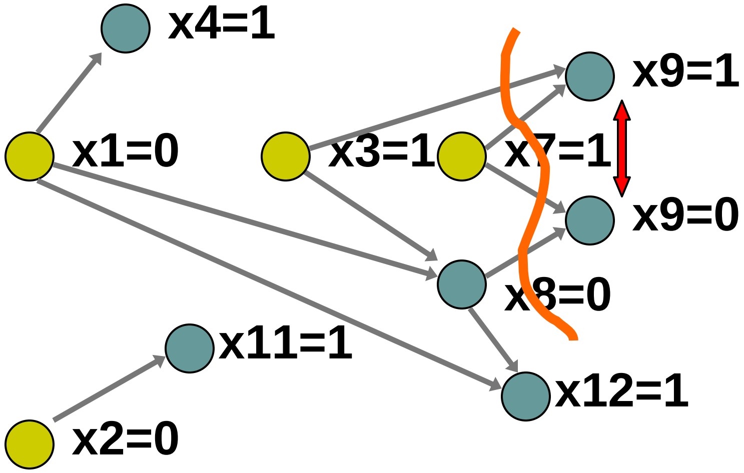
$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow$ conflict



Conflict Driven Learning 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’





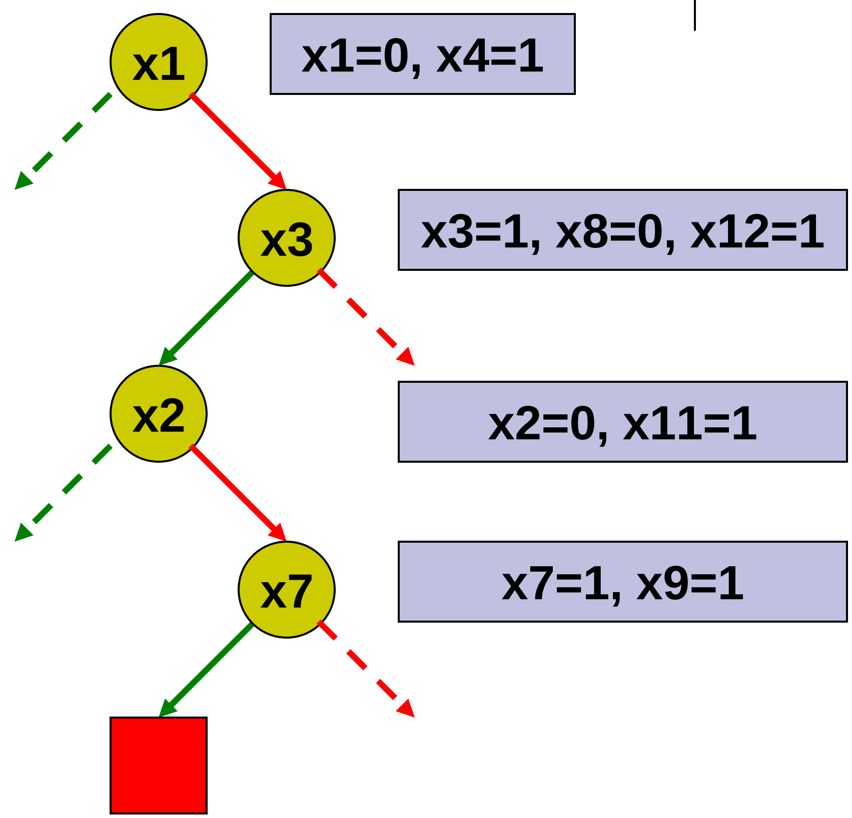
$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow$ conflict

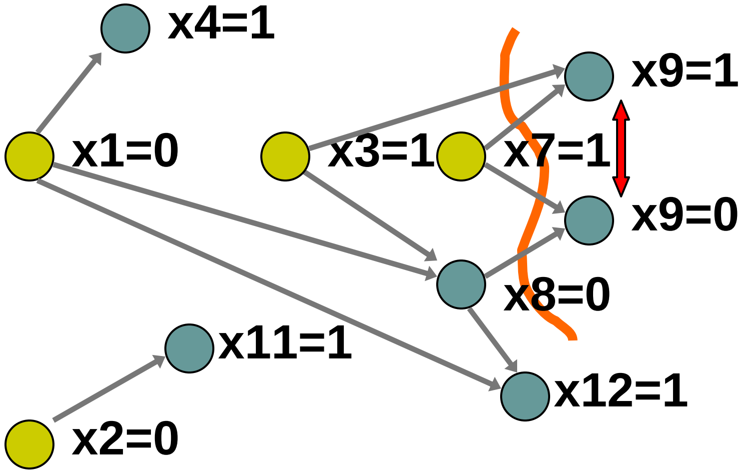
Add conflict clause: x3’+x7’+x8

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'

x3'+x7'+x8

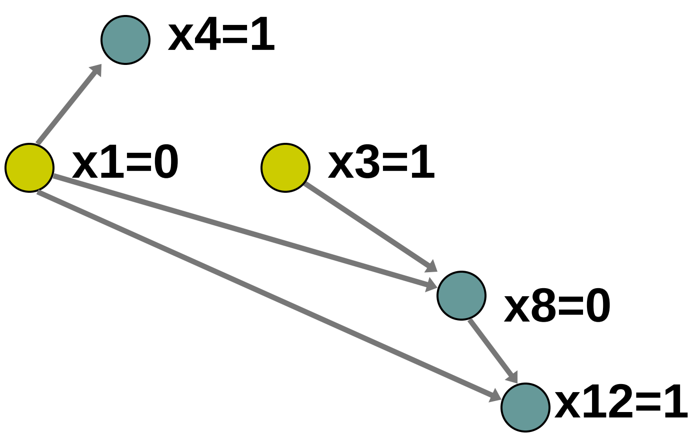


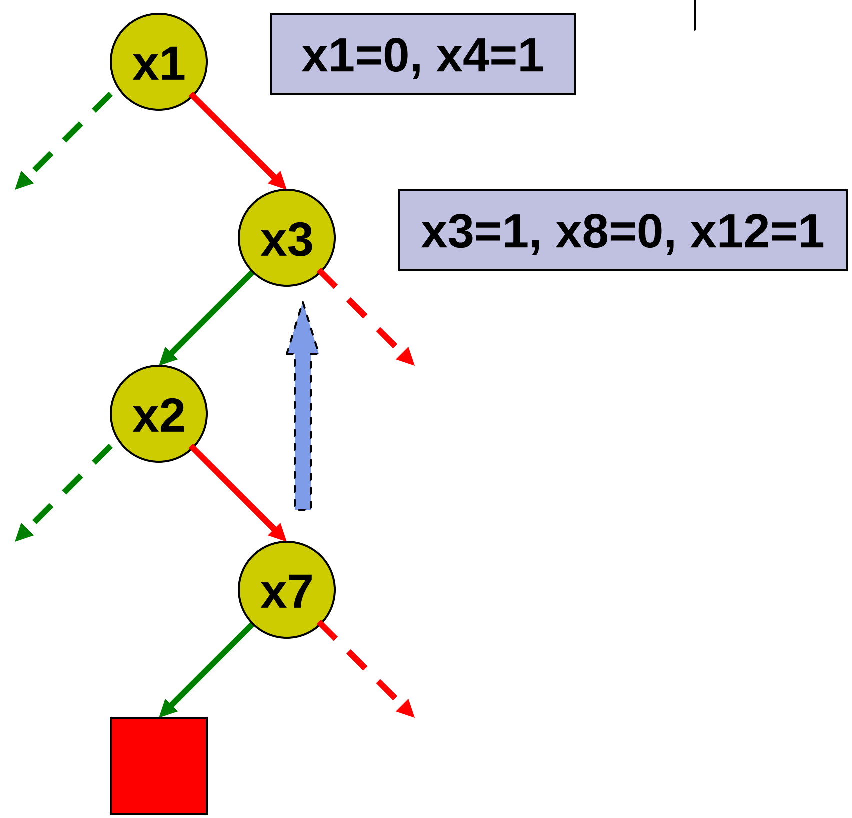


x3=1∧x7=1∧x8=0 → conflict
Add conflict clause: x3'+x7'+x8

Conflict Driven Learning and Non-chronological Backtracking

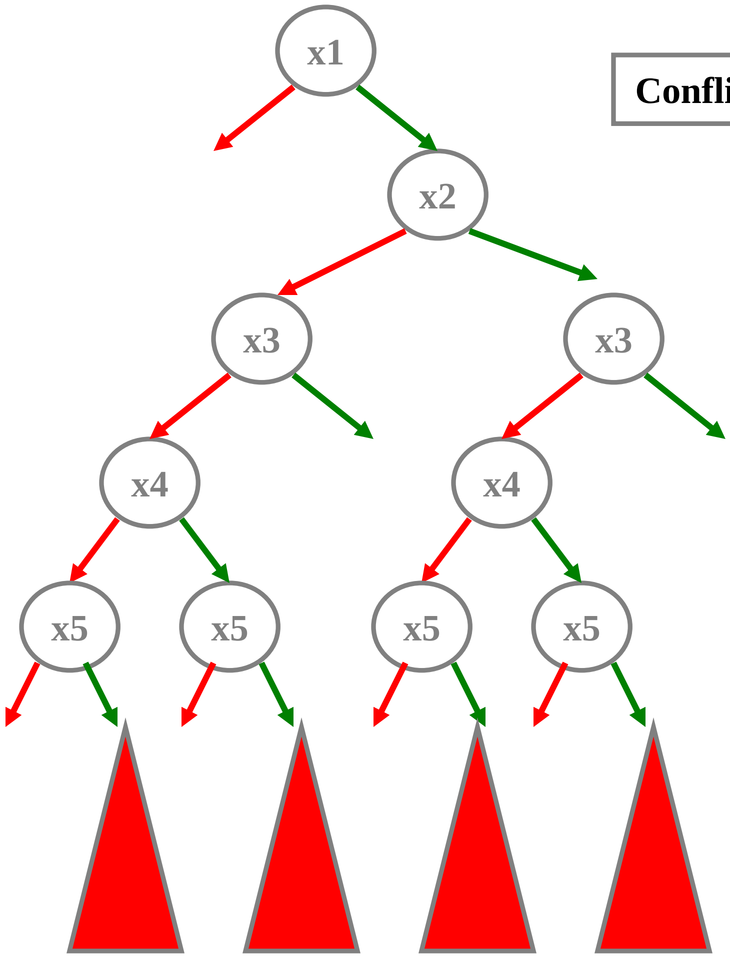
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'
x3' + x8 + x7'





Backtrack to the decision level of x3=1
With implication x7 = 0

What’s the big deal?

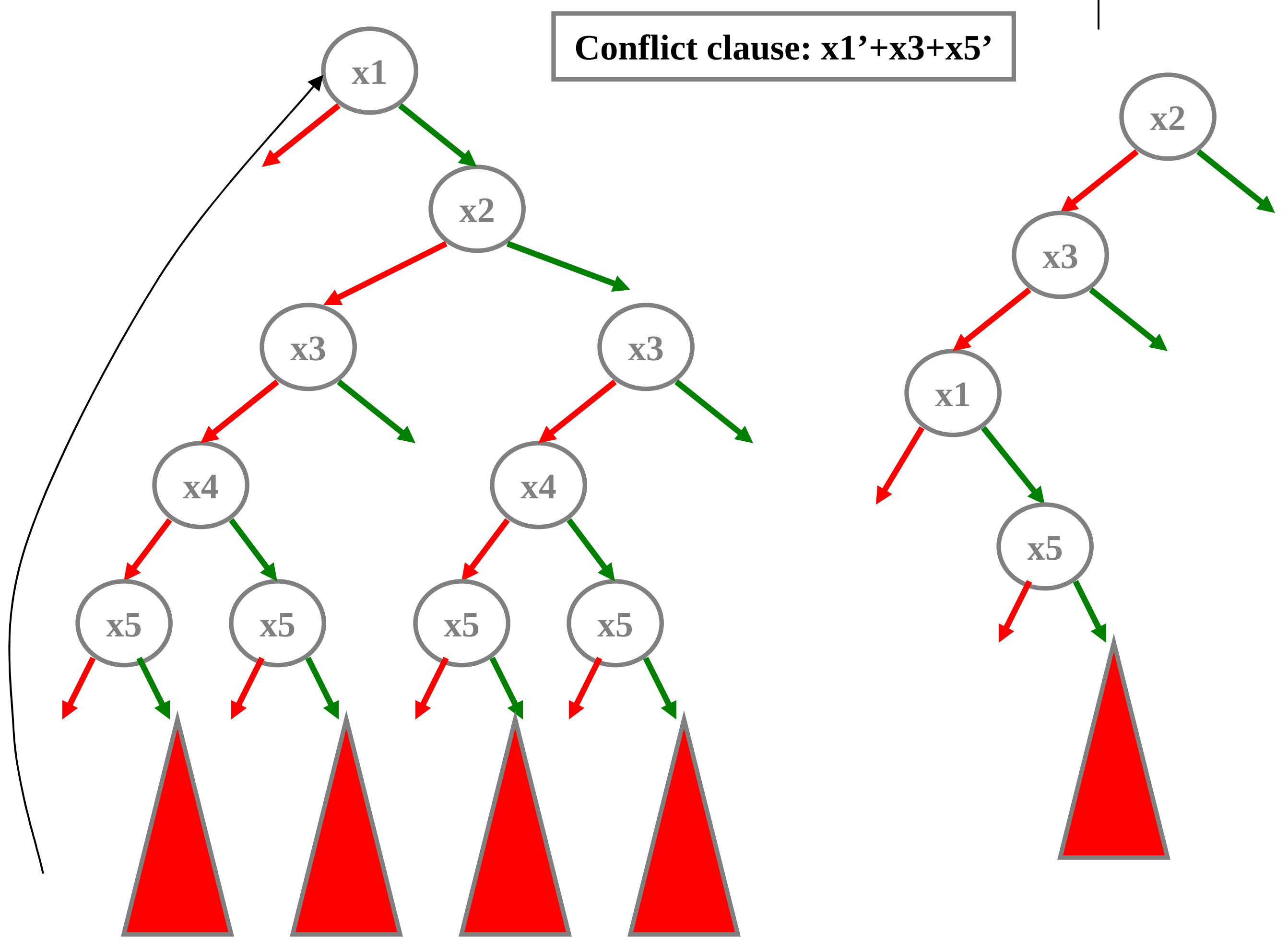


Significantly prune the search space – learned clause is useful forever!

Useful in generating future conflict clauses.

Restart

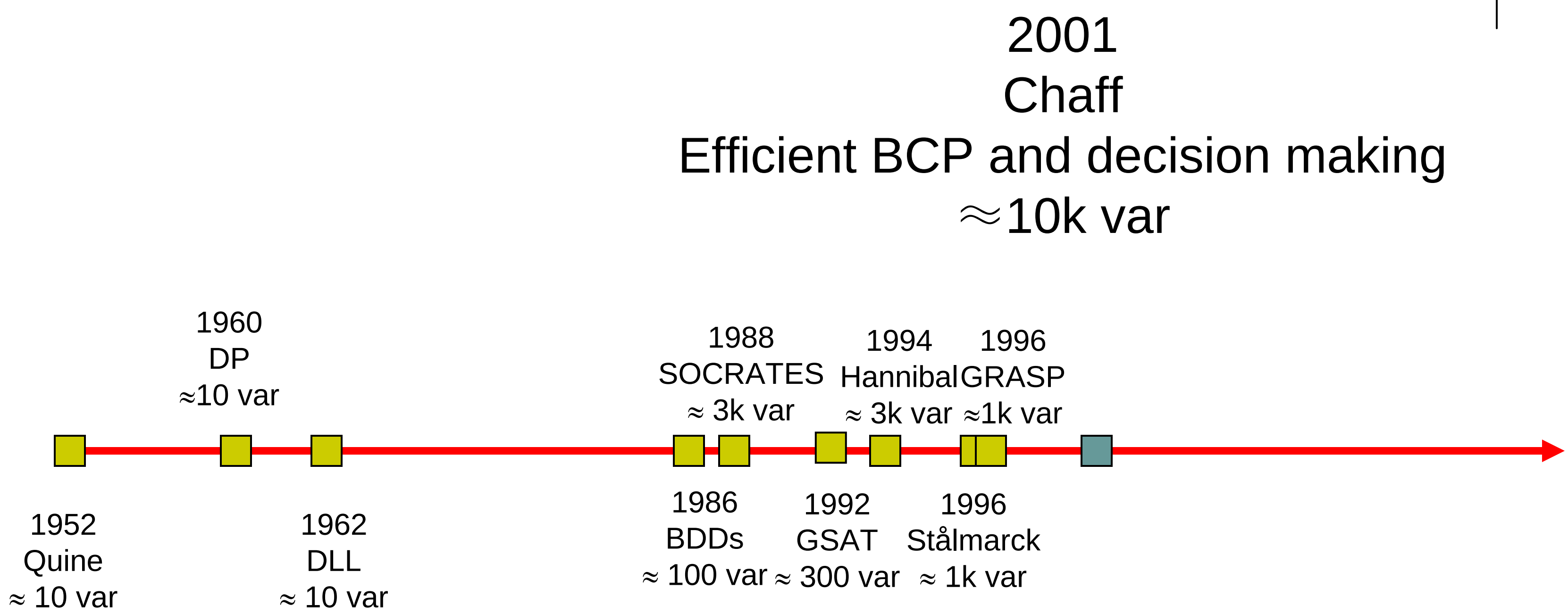
- Abandon the current search tree and reconstruct a new one
- Helps reduce variance - adds to robustness in the solver
- The clauses learned prior to the restart are *still there* after the restart and can help pruning the search space



SAT becomes practical!

- Conflict driven learning greatly increases the capacity of SAT solvers (several thousand variables) for structured problems
- Realistic applications became plausible
  - 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

The Timeline



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. (43 citations)
- Widely Used:
  - Formal verification
    - Hardware and software
  - 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
  - Alloy – Software Model Analyzer at M.I.T.
  - haRVey – Refutation-based first-order logic theorem prover
  - Several industrial users – Intel, IBM, Microsoft, …

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 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

Recognition that this is as much a large (in-memory) database problem as it is a search problem.

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

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

	zChaff	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

BCP Algorithm (1/8)

- What “causes” an implication? When can it occur?
  - All literals in a clause but one are assigned to 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 have been assigned to False
  - 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: (v1 + v2 + v3 + v4 + v5)
    - (**v1=X + v2=X** + v3=? {i.e. X or 0 or 1} + v4=? + v5=?)

BCP Algorithm (1.1/8)

- Big Invariants
  - Each clause has two watched literals.
  - If a clause can become unit 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)
    - (**v1=X** + **v2=X** + v3=? + v4=? + v5=?)
- BCP consists of identifying unit (and conflict) clauses (and the associated implications) while maintaining the “Big Invariants”

BCP Algorithm (2/8)

- Let’s illustrate this with an example:

```
v2 + v3 + v1 + v4 + v5
v1 + v2 + v3′
v1 + v2′
v1′+ v4
v1′
```

BCP Algorithm (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 Algorithm (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 Algorithm (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 Algorithm (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 unit.

BCP Algorithm (3.3/8)

- 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
```

- 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 unit.
- We *certainly* need not process any clauses where neither watched literal changes state (in this example, where v1 is not watched).

BCP Algorithm (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 Algorithm (4.1/8)

- 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:
```

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

```
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 Algorithm (4.2/8)

- 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:
```

```
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 unit. We record the new implication of v2’, and add it to the queue of assignments to process. Since the clause cannot again become unit, our invariants are maintained.

BCP Algorithm (5/8)

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

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

State:(v1=F, v2=F)
Pending:
```

→

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

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 unit. We record the new implication of v3’, and add it to the queue of assignments to process. Since the clause cannot again become unit, our invariants are maintained.

BCP Algorithm (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 Algorithm (7/8)

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

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

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

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

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.

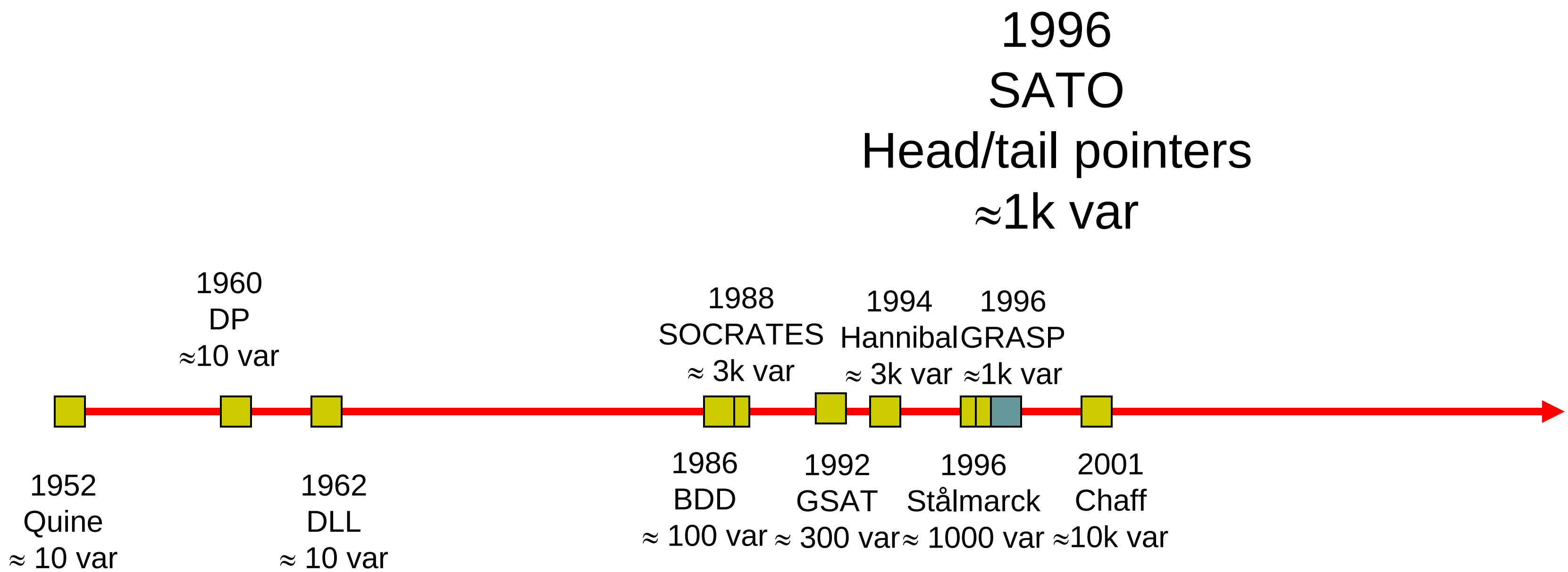
BCP Algorithm (8/8)

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

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

- The first clause is already satisfied by v4 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 instance is SAT, and we are done.

The Timeline



SATO

H. Zhang, M. Stickel, “An efficient algorithm for unit-propagation” *Proc. of the Fourth International Symposium on Artificial Intelligence and Mathematics,* 1996. (7 citations)

H. Zhang, “SATO: An Efficient Propositional Prover” *Proc. of International Conference on Automated Deduction*, 1997. (63 citations)

- The Invariants
  - Each clause has a head pointer and a tail pointer.
  - All literals in a clause before the head pointer and after the tail pointer have been assigned false.
  - If a clause can become unit via any sequence of assignments, then this sequence will include an assignment to one of the literals pointed to by the head/tail pointer.

Chaff vs. SATO: A Comparison of BCP

Chaff: v1 + v2’ + v4 + v5 + v8’ + v10 + v12 + v15

SATO: v1 + v2’ + v4 + v5 + v8’ + v10 + v12 + v15

Chaff vs. SATO: A Comparison of BCP

Chaff: v1 + v2’ + v4 + v5 + v8’ + v10 + v12 + v15

SATO: v1 + v2’ + v4 + v5 + v8’ + v10 + v12 + v15

Chaff vs. SATO: A Comparison of BCP

Chaff: v1 + v2' + v4 + v5 + v8' + v10 + v12 + v15

SATO: v1 + v2' + v4 + v5 + v8' + v10 + v12 + v15

Chaff vs. SATO: A Comparison of BCP

Chaff: v1 + v2’ + v4 + v5 + v8’ + v10 + v12 + v15

SATO: v1 + v2’ + v4 + v5 + v8’ + v10 + v12 + v15

Chaff vs. SATO: A Comparison of BCP

Chaff: v1 + v2’ + v4 + v5 + v8’ + v10 + v12 + v15

Implication

SATO: v1 + v2’ + v4 + v5 + v8’ + v10 + v12 + v15

Chaff vs. SATO: A Comparison of BCP

Chaff: v1 + v2' + v4 + v5 + v8' + v10 + v12 + v15

SATO: v1 + v2' + v4 + v5 + v8' + v10 + v12 + v15

Chaff vs. SATO: A Comparison of BCP

Chaff: **v1** + **v2'** + v4 + v5 + **v8'** + v10 + v12 + **v15**

Backtrack in Chaff

SATO: **v1** + **v2'** + **v4** + **v5** + **v8'** + **v10** + **v12** + **v15**

Chaff vs. SATO: A Comparison of BCP

Chaff: v1 + v2’ + v4 + v5 + v8’ + v10 + v12 + v15

Backtrack in SATO

SATO: v1 + v2’ + v4 + v5 + v8’ + v10 + v12 + v15

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

Decision Heuristics – Conventional Wisdom

- DLIS (Dynamic Largest Individual Sum) 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→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. (8 citations)

Chaff Decision Heuristic - VSIDS

- Variable State Independent Decaying Sum
  - Rank variables by literal count in the initial clause database
  - Only increment counts as new clauses are added.
  - Periodically, divide all counts by a constant.
- Quasi-static:
  - Static because it doesn't depend on variable state
  - Not static because it gradually changes as new clauses are added
    - Decay causes bias toward *recent* conflicts.
- Use heap to find unassigned variable with the highest ranking
    - Even single linear pass though 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

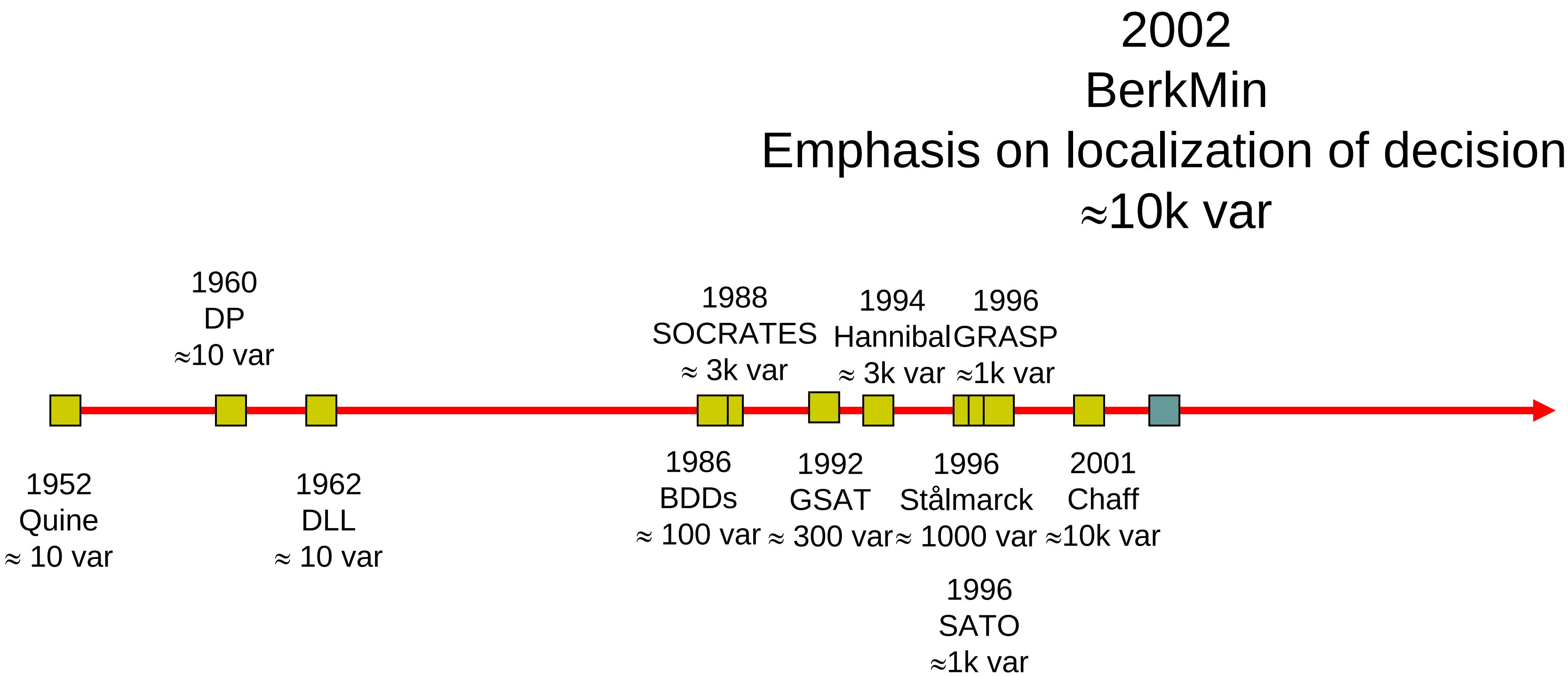
Interplay of BCP and the Decision Heuristic

- 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 v2=T, v7=F, v9=T, v1=T (and any implications thereof)
  - Then a conflict is encountered that forces v2=F
    - The next decisions may still be v7=F, v9=T, v1=T !
      - VSIDS variable ranks change slowly…
    - But the BCP engine has recently processed these assignments …
      - so these variables are unlikely to still be watched.
- In a more general sense, the more “active” a variable is, the more likely it is to *not* be watched.

Interplay of Learning and the Decision Heuristic

- Again, this is an intuitive description …
- Learnt clauses capture relationships between variables
- Learnt clauses bias decision strategy to a smaller set of variables through decision heuristics like VSIDS
  - Important when there are 100k variables!
- Decision heuristic influences which variables appear in learnt clauses
  - Decisions →implications →conflicts →learnt clause
- Important for decisions to keep search strongly localized

The Timeline



Berkmin – Decision Making Heuristics

E. Goldberg, and Y. Novikov, “BerkMin: A Fast and Robust Sat-Solver”, *Proc. DATE* 2002, pp. 142-149. (5 citations)

- Identify the most recently learned clause which is unsatisfied
- Pick most active variable in this clause to branch on
- Variable activities
  - updated during conflict analysis
  - decay periodically
- If all learnt conflict clauses are satisfied, choose variable using a global heuristic
- Increased emphasis on “locality” of decisions

SAT Solver Competition!

SAT03 Competition

http://www.lri.fr/~simon/contest03/results/mainlive.php

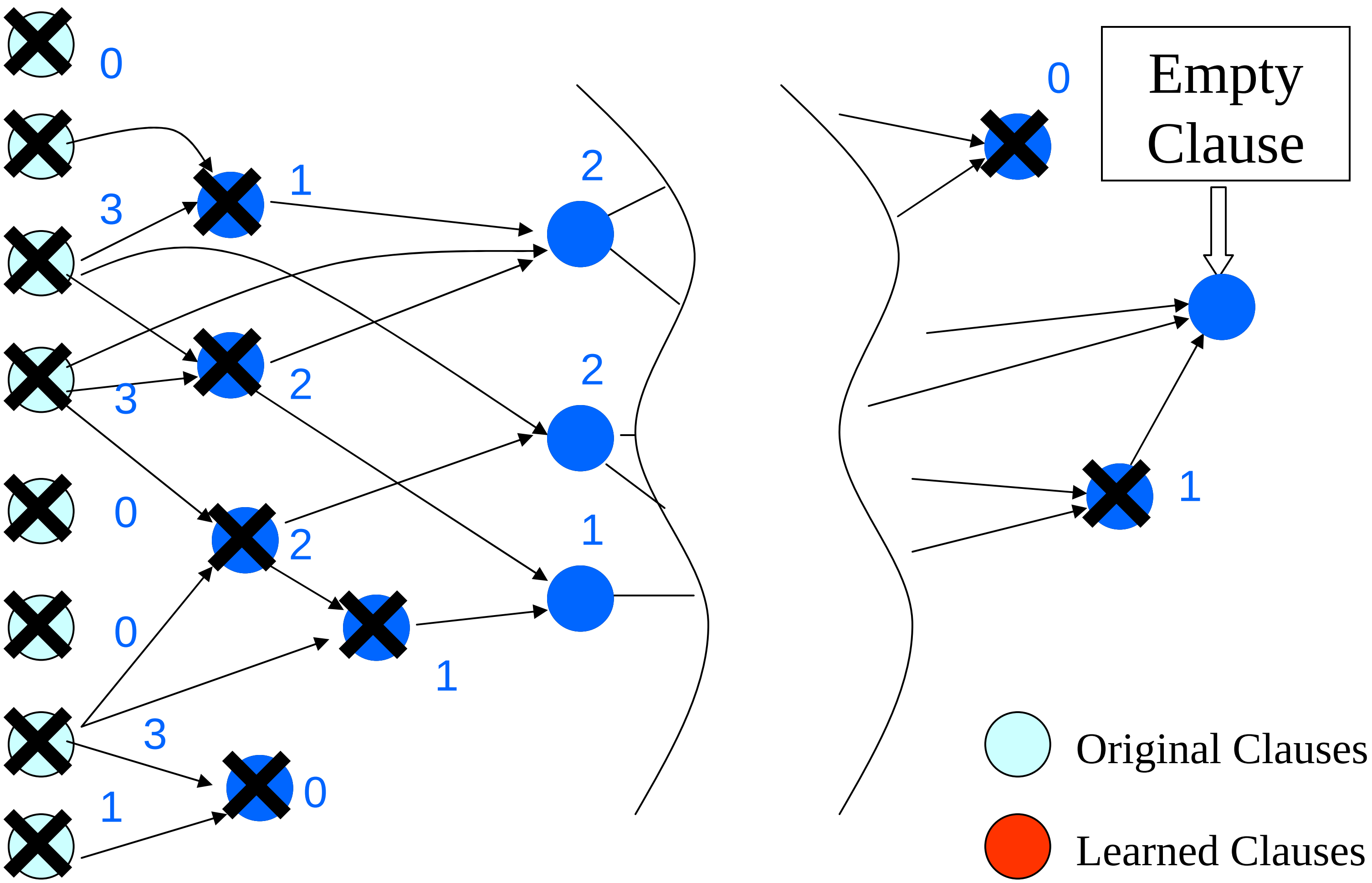
34 solvers, 330 CPU days, 1000s of benchmarks

SAT04 Competition is going on right now …

Certifying a SAT Solver

- Do you trust your SAT solver?
  - If it claims the instance is satisfiable, it is easy to check the claim.
  - How about unsatisfiable claims?
- Search process is actually a proof of unsatisfiability by resolution
  - Effectively a series of resolutions that generates an empty clause at the end
- Need an independent check for this proof
- Must be automatic
  - Must be able to work with current state-of-the-art SAT solvers
- The SAT solver dumps a trace (on disk) during the solving process from which the resolution graph can be derived
- A third party checker constructs the empty clause by resolution using the trace

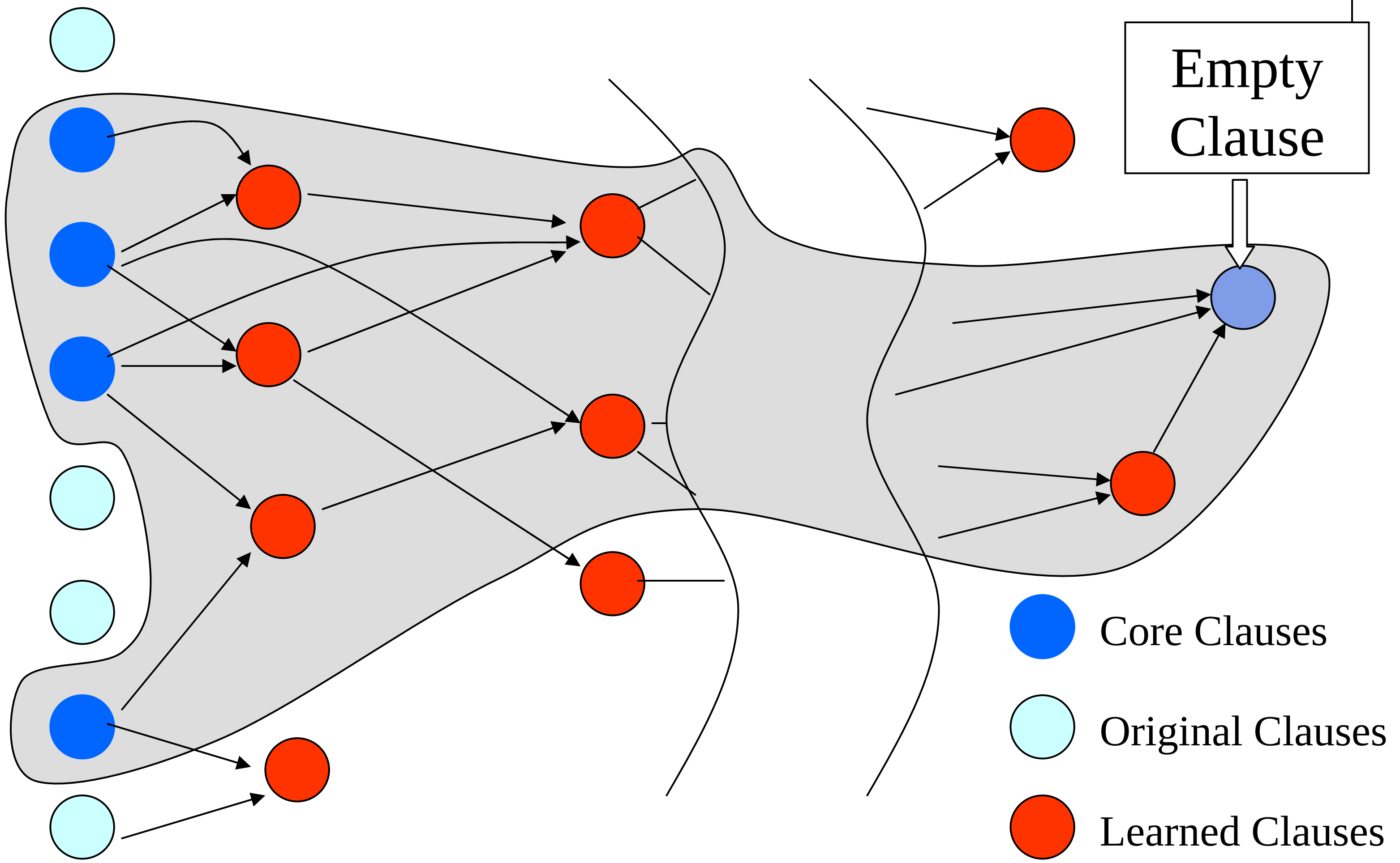
A Disk-Based BFS Algorithm



Extracting an Unsatisfiable Core

- Extract a small subset of unsatisfiable clauses from an unsatisfiable SAT instance
- Motivation:
  - Debugging and redesign: SAT instances are often generated from real world applications with certain expected results:
    - If the expected result is unsatisfiable, but the instance is satisfiable, then the solution is a “stimulus” or “input vector” or “counter-example” for debugging
      - Combinational Equivalence Checking
      - Bounded Model Checking
    - What if the expected result is satisfiable?
      - SAT Planning
      - FPGA Routing
  - Relaxing constraints:
    - If several constraints make a safety property hold, are there any redundant constraints in the system that can be removed without violating the safety property?

The Core as a Checker By-Product



- Can do this iteratively
- Can result in very small cores

Summary

- Rich history of emphasis on practical efficiency.
- Presence of drivers results in maximum progress.
- 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.