DLL Algorithm

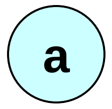
- Davis, Logemann and Loveland
- Basic framework for many modern SAT solvers
- Also known as DPLL for historical reasons

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)

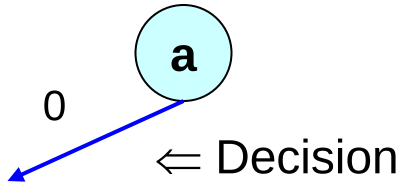
Basic DLL Procedure - DFS

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



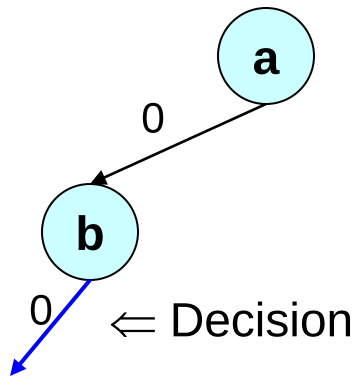
Basic DLL Procedure - DFS

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



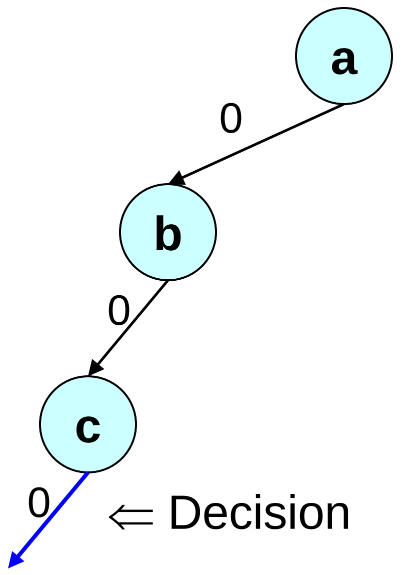
Basic DLL Procedure - DFS

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



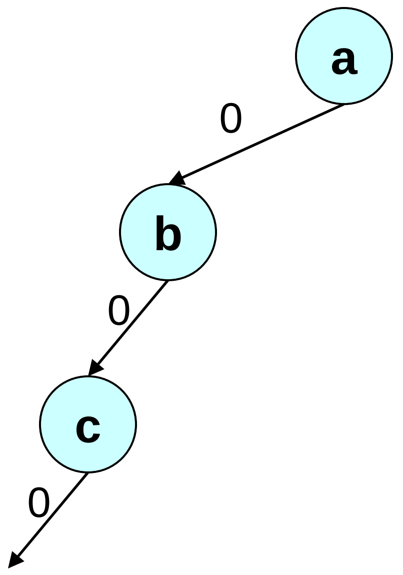
Basic DLL Procedure - DFS

(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)



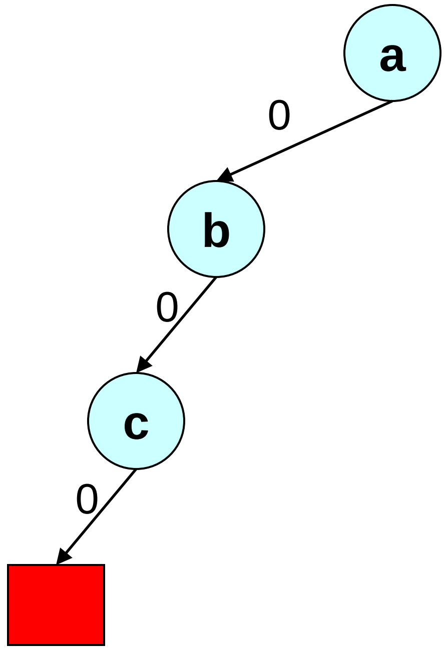
Implication Graph

$$\begin{tikzcd}
a=0 \arrow[r, "(a + c + d)"] \arrow[dr] & d=1 \arrow[d, leftrightarrow] \\
c=0 \arrow[r, "(a + c + d')"'] \arrow[ur] & d=0
\end{tikzcd}$$

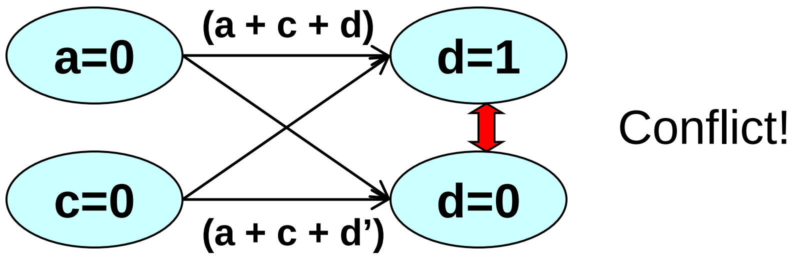
Conflict!

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)

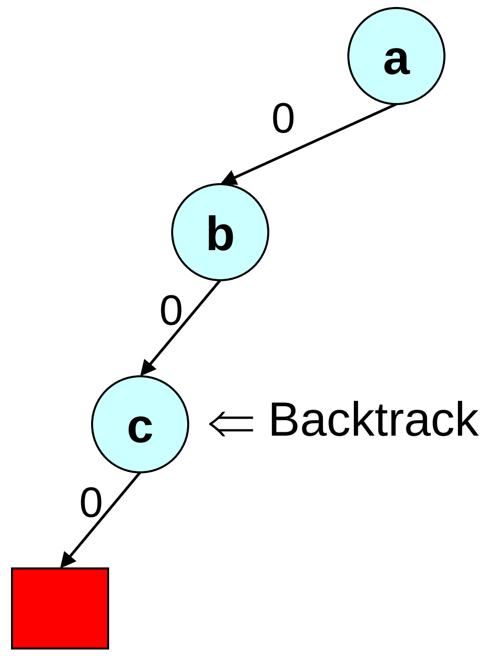


Implication Graph



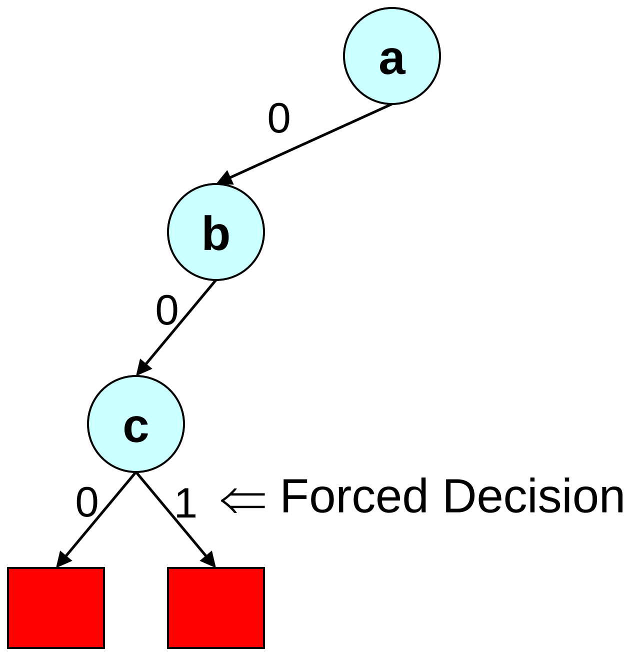
Basic DLL Procedure - DFS

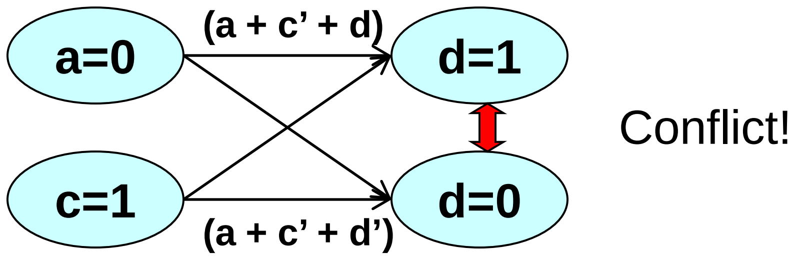
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

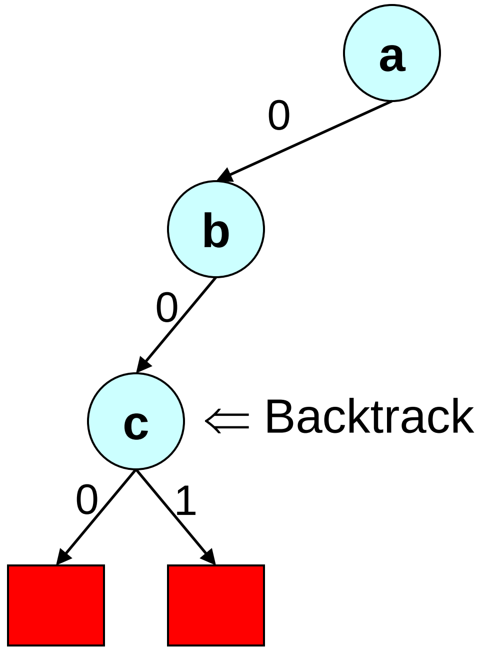
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)





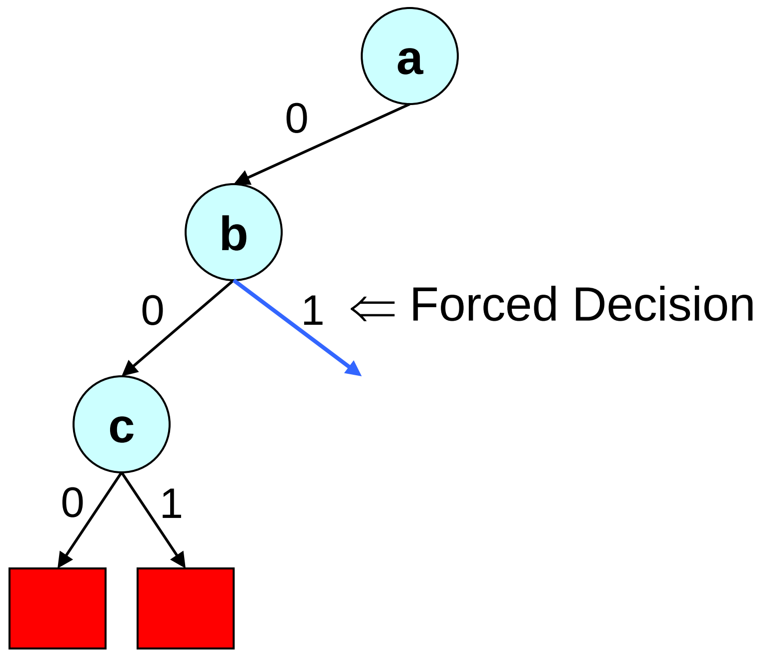
Basic DLL Procedure - DFS

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



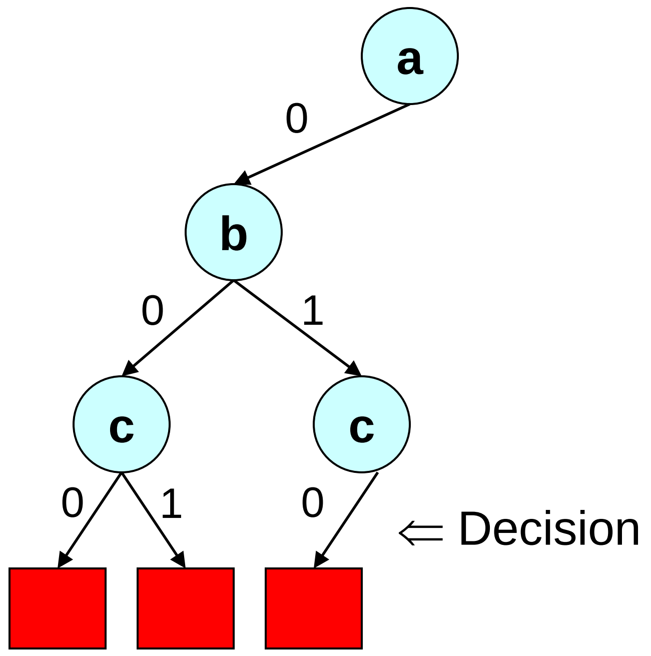
Basic DLL Procedure - DFS

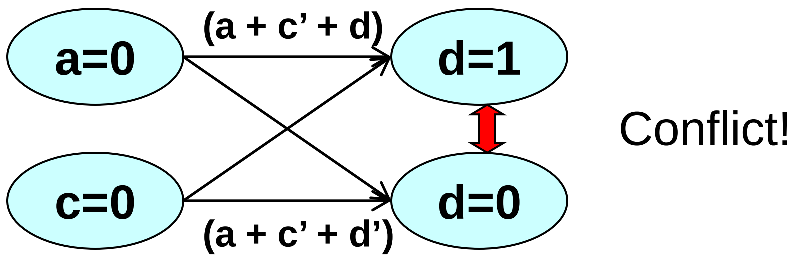
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

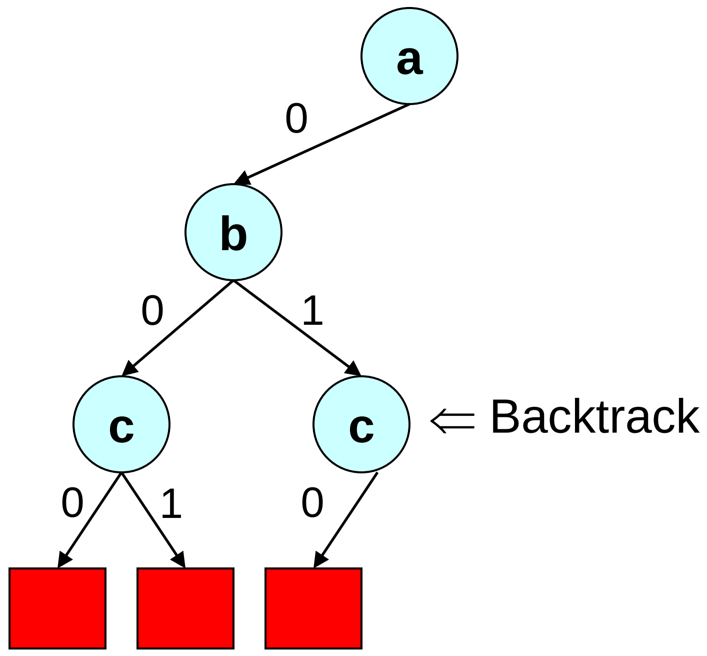
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)





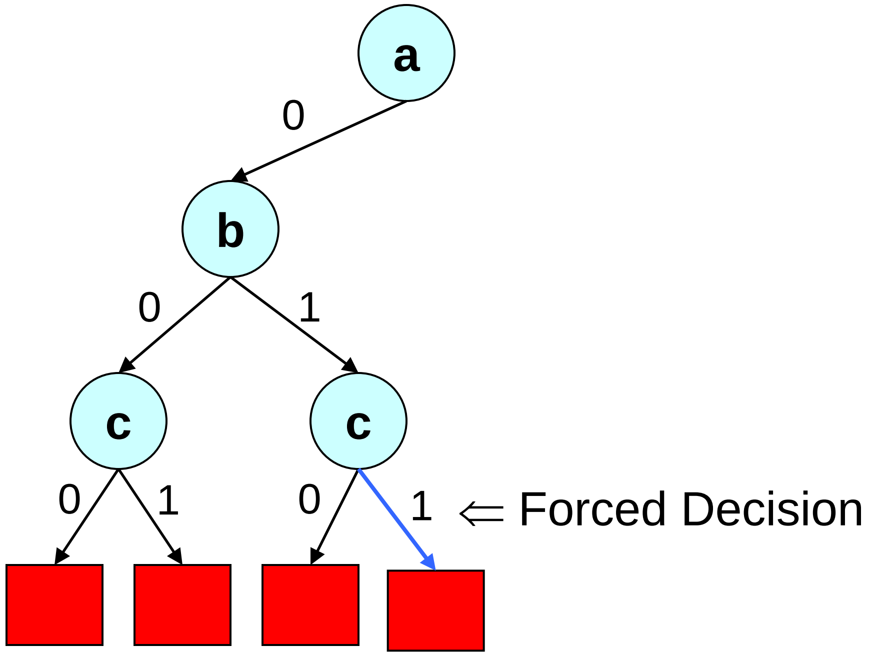
Basic DLL Procedure - DFS

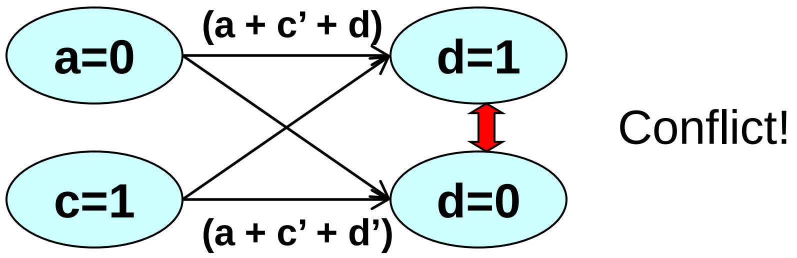
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

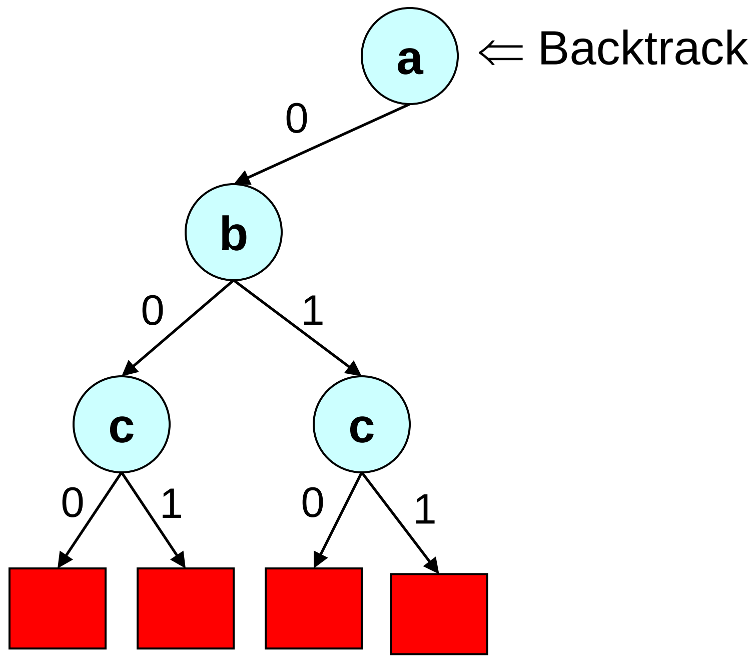
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)





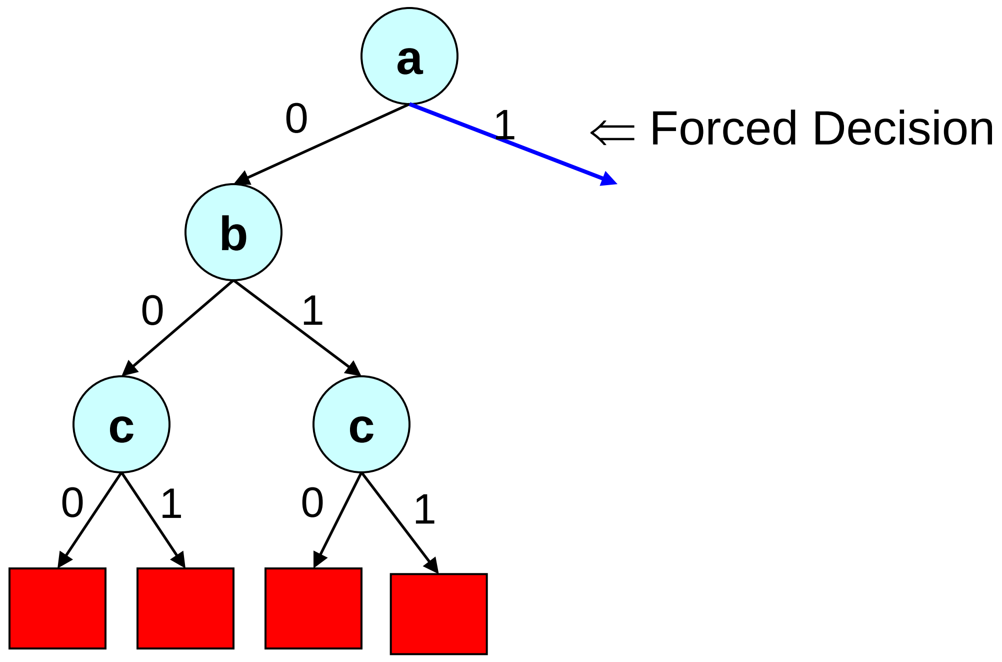
Basic DLL Procedure - DFS

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



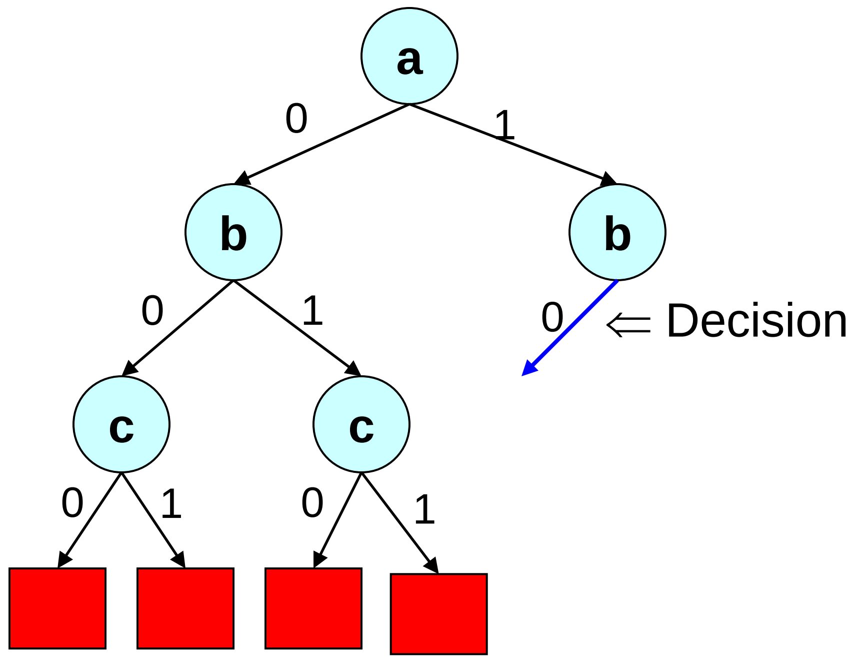
Basic DLL Procedure - DFS

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



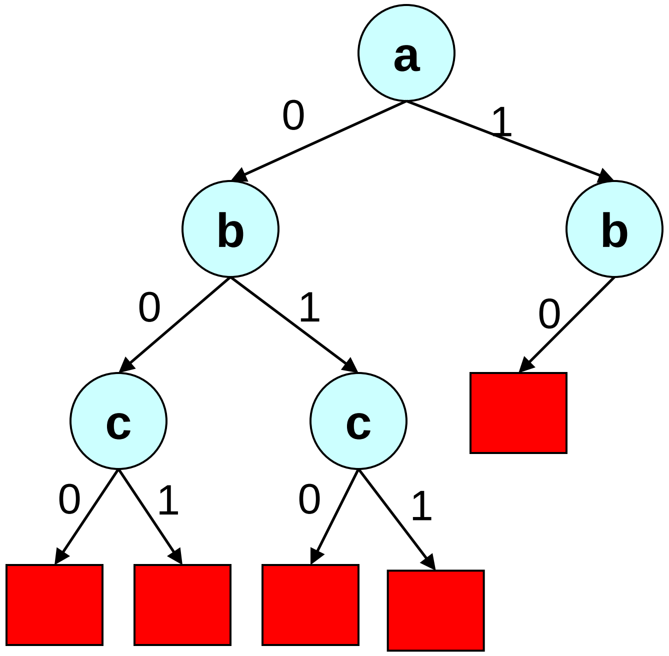
Basic DLL Procedure - DFS

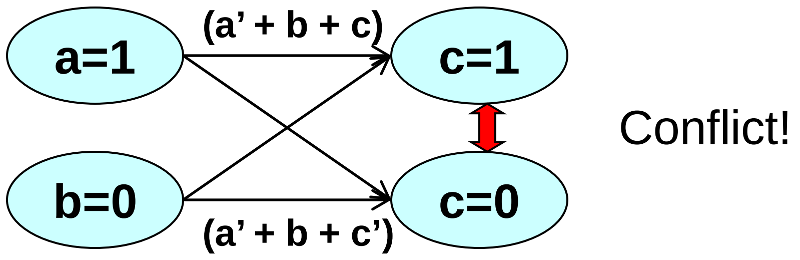
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure - DFS

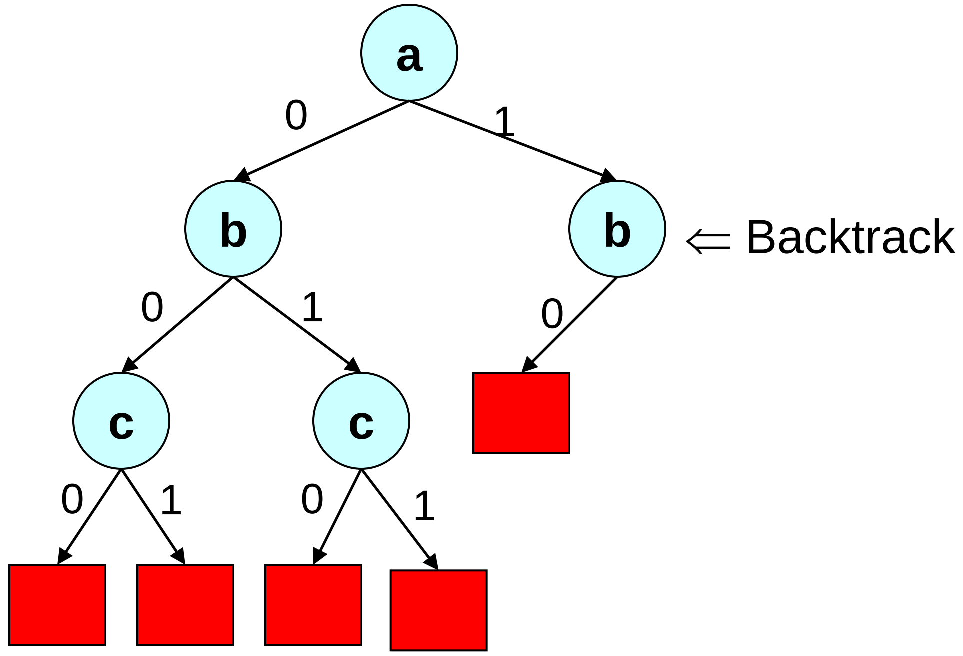
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)





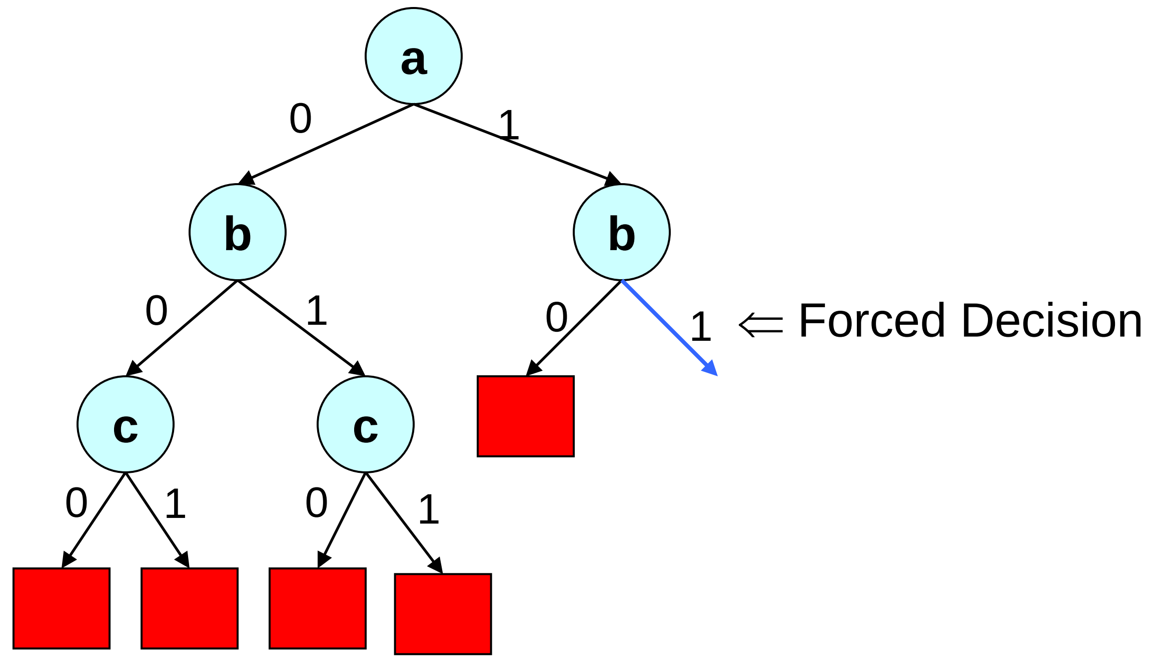
Basic DLL Procedure - DFS

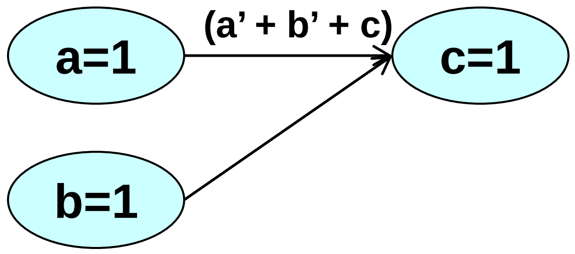
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



Basic DLL Procedure - DFS

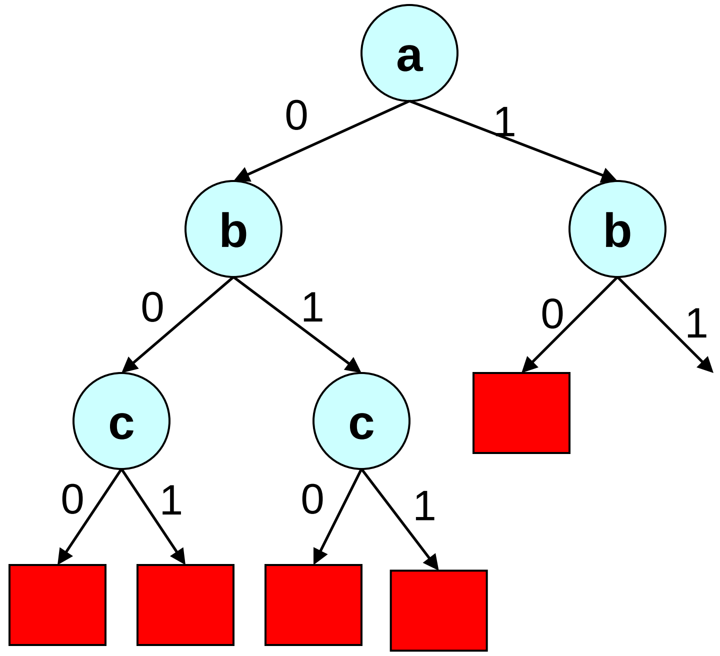
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

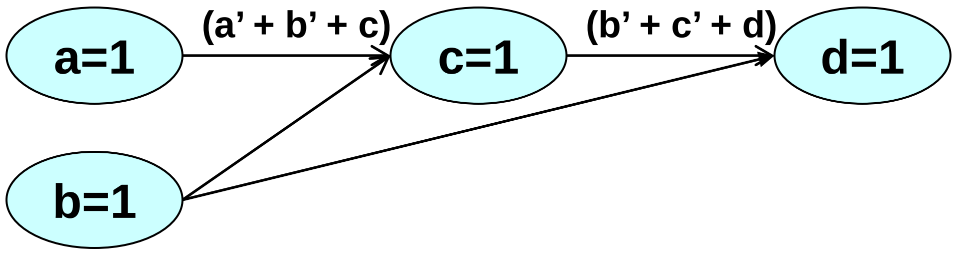




Basic DLL Procedure - DFS

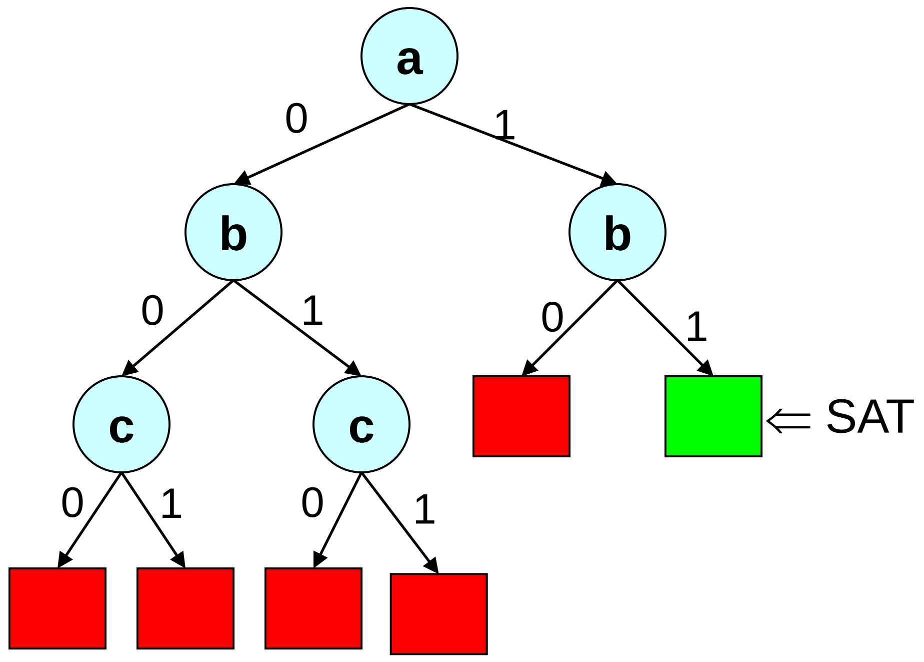
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

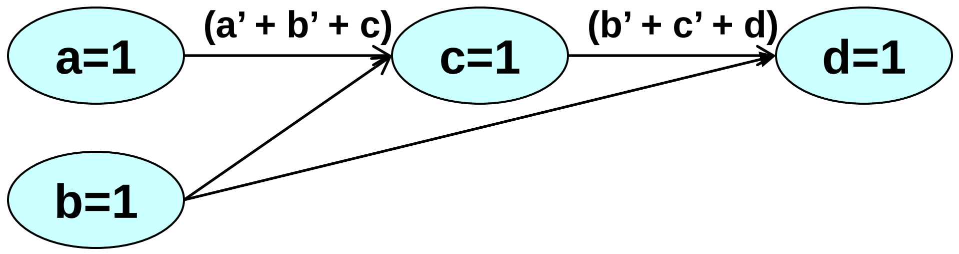




Basic DLL Procedure - DFS

(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.
- Workhorse of DLL based algorithms – used to reduce the number of decisions made

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 (partly due to capacity of computers at the time)

Exercise

- Apply the DLL algorithm without and with the use of implications to determine the satisfiability of the following formula:

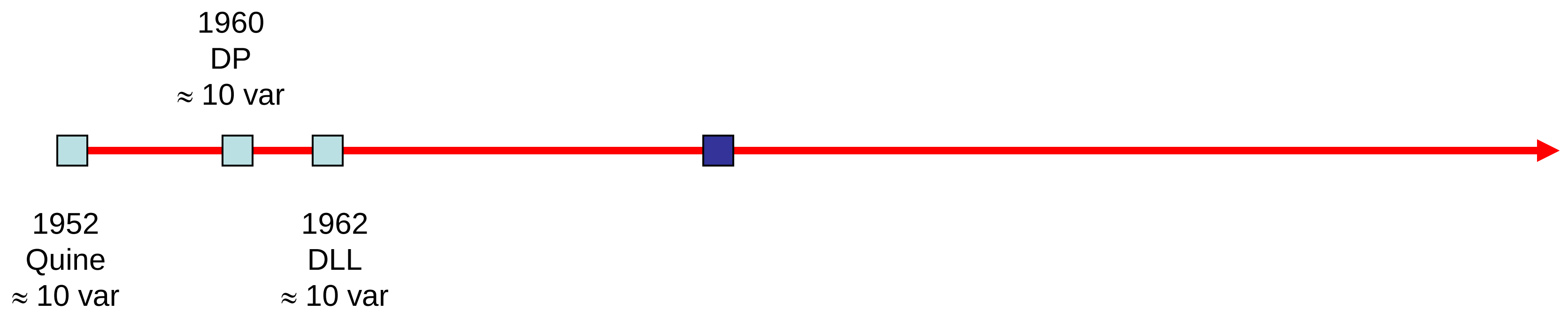
$$(a+b+c)(a+b+c')(a'+b+c')(a+c+d)(a'+c+d)(a'+c+d')(b'+c'+d')(b'+c'+d)$$

General Principles

- Backtracking can be an effective approach to solving search problems by systematically exploring the solution space
- Use of implications significantly reduces search space
  - No need to backtrack on decisions that are forced by implications

The Timeline

1986
Binary Decision Diagrams (BDDs)
≈100 var



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

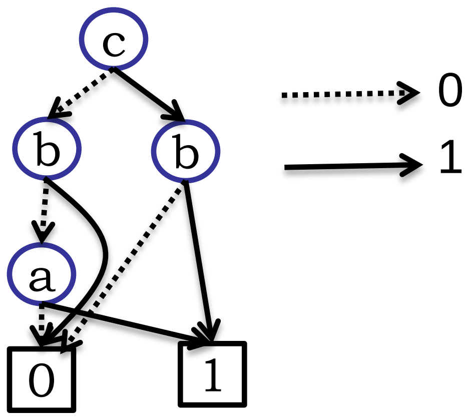
Binary Decision Diagram (BDD)

- Binary Decision Tree has large number of nodes
- Key idea: Share sub-trees and eliminate redundancy to reduce size
- More about BDDs later in the class

Example:

a b c	f
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	1
1 0 1	0
1 1 0	0
1 1 1	1

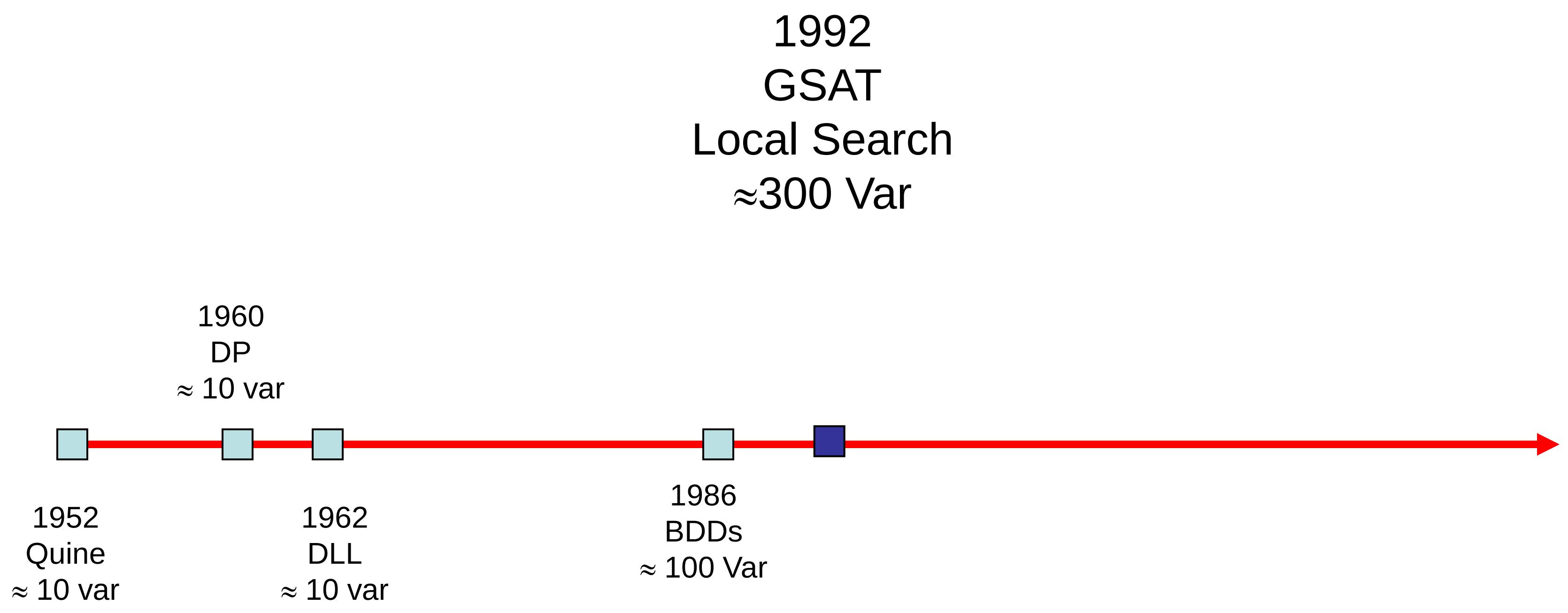
Binary Decision Diagram:



Using BDDs to Solve SAT

- Store the formula in a Reduced Ordered BDD (ROBDD) representation.
  - Compacted form of the binary decision tree.
- Construction rules guarantee canonicity
  - Only one way to represent constant 0 (unsatisfiable) function!
  - Any other function is satisfiable
- Overkill for SAT.

The Timeline

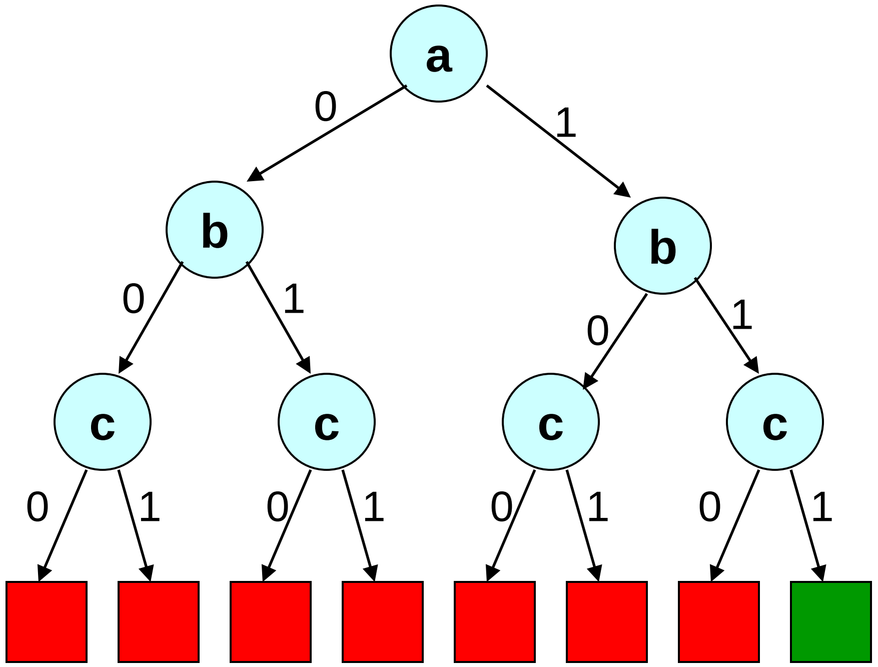


B. Selman, H. Levesque, and D. Mitchell. “A new method for solving hard satisfiability problems,” *Proc. National Conference on Artificial Intelligence (AAAI)*, 1992.

B. Selman and H. Kautz “Noise Strategies for Improving Local Search,” *Proc. National Conference on Artificial Intelligence (AAAI)*, 1994.

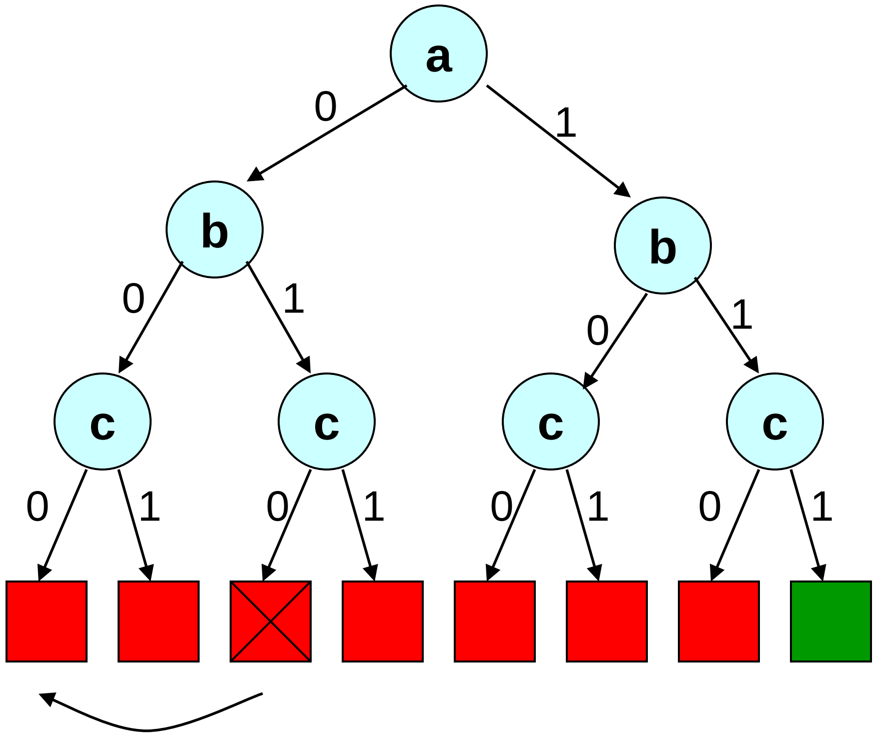
Local Search

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



Local Search

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



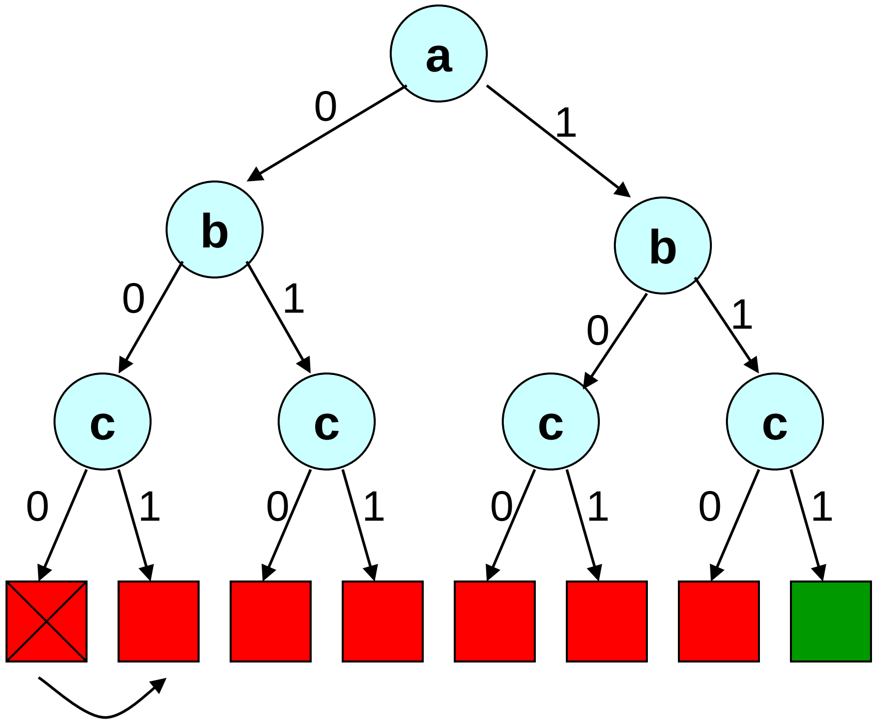
Cost Function (CF):

# of un-satisfied clauses:

CF(a=0,b=1,c=0) = 1

Local Search

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



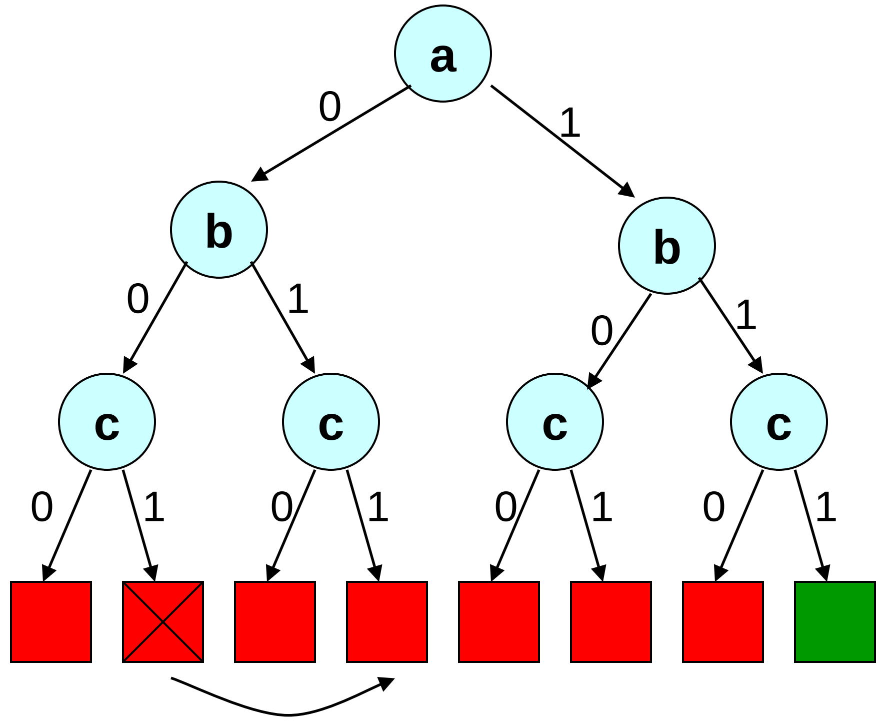
Cost Function F:

# of un-satisfied clauses:

CF(a=0,b=0,c=0) = 2

Local Search

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



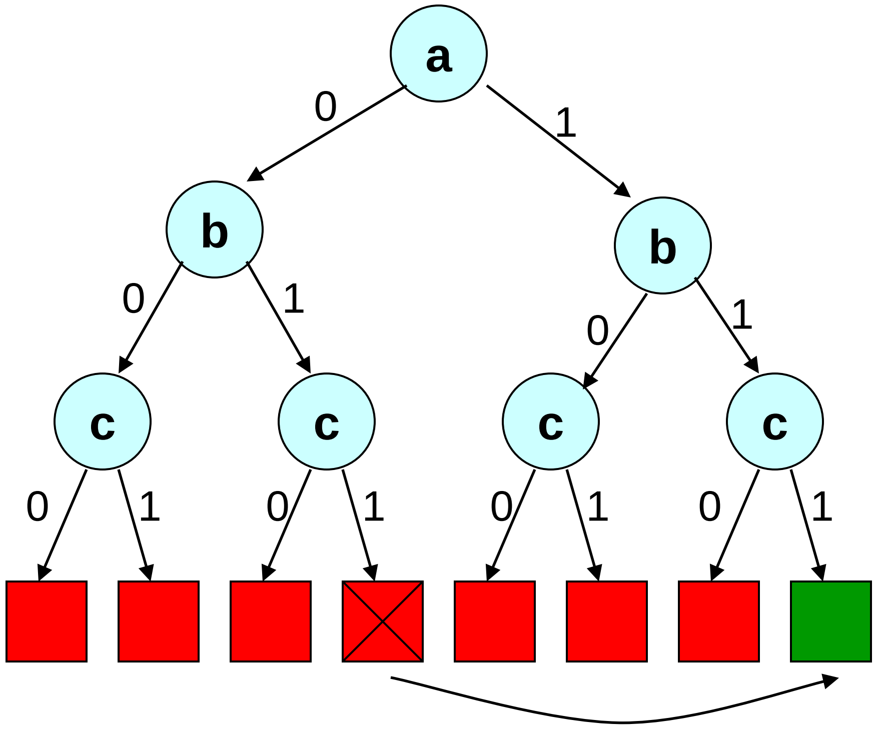
Cost Function F:

# of un-satisfied clauses:

CF(a=0,b=0,c=1) = 1

Local Search

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



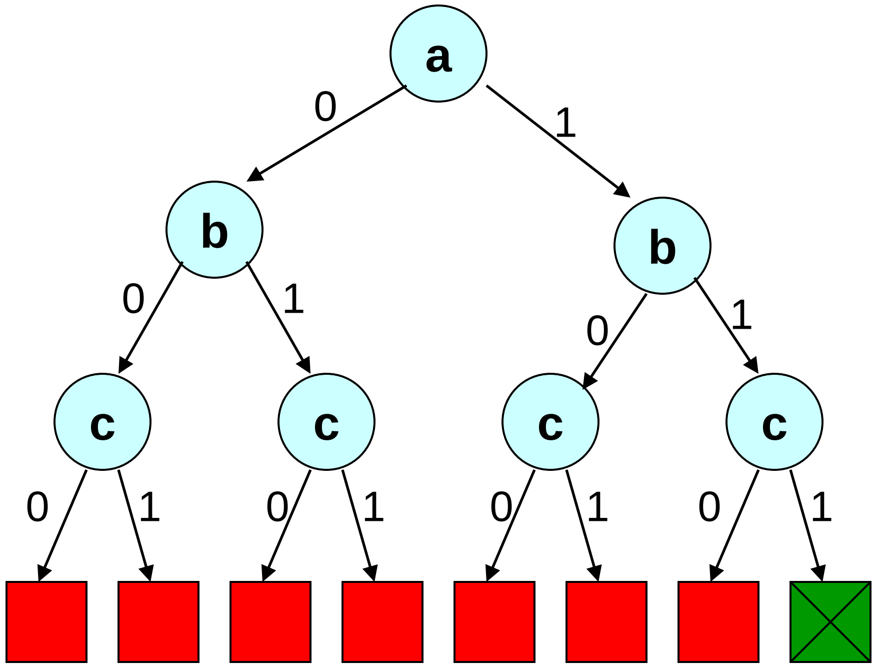
Cost Function F:

# of un-satisfied clauses:

CF(a=0,b=1,c=1) = 1

Local Search

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



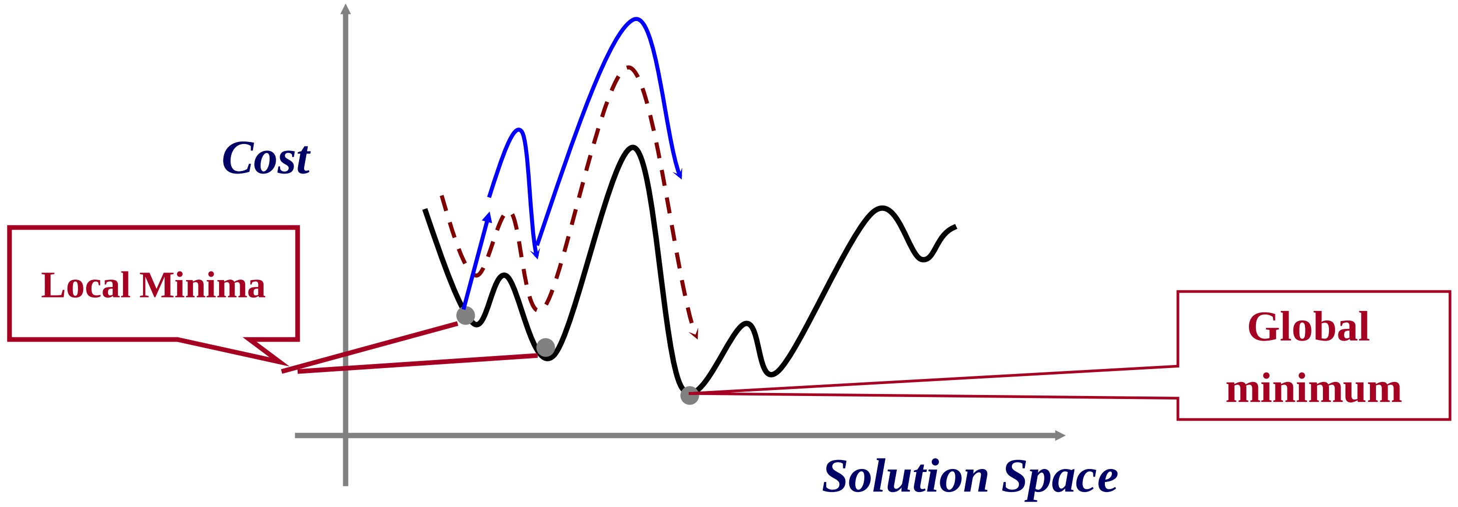
Cost Function F:

# of un-satisfied clauses:

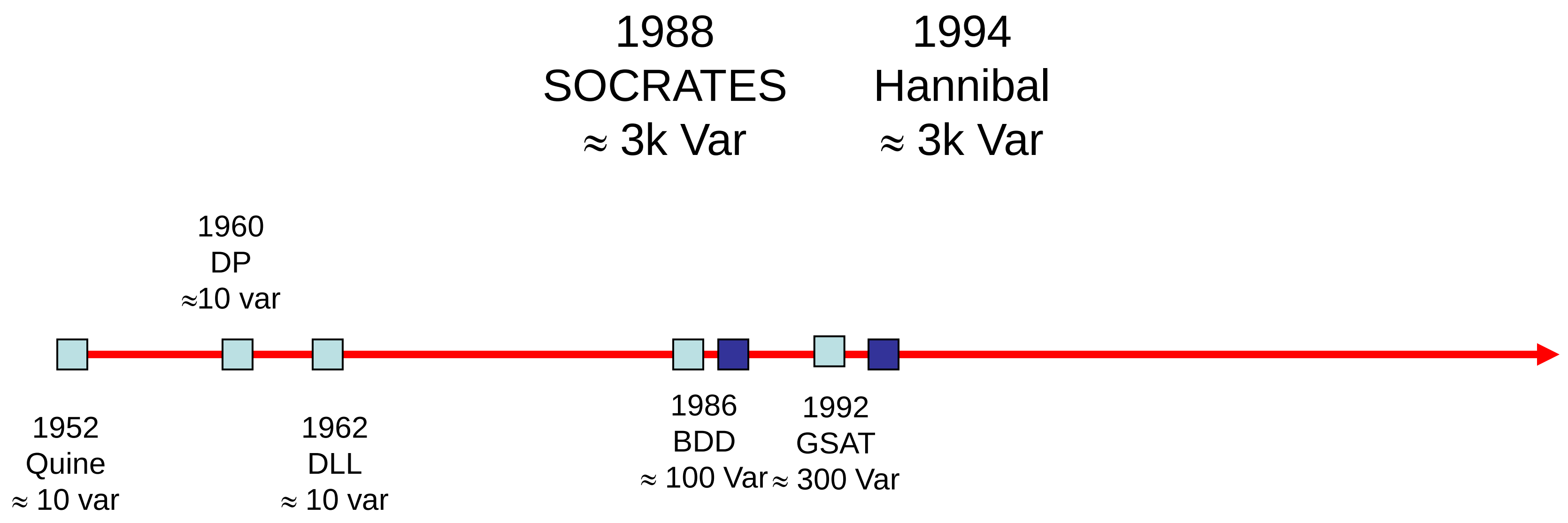
CF(a=1,b=1,c=1) = 0

Local Search (GSAT, WSAT)

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



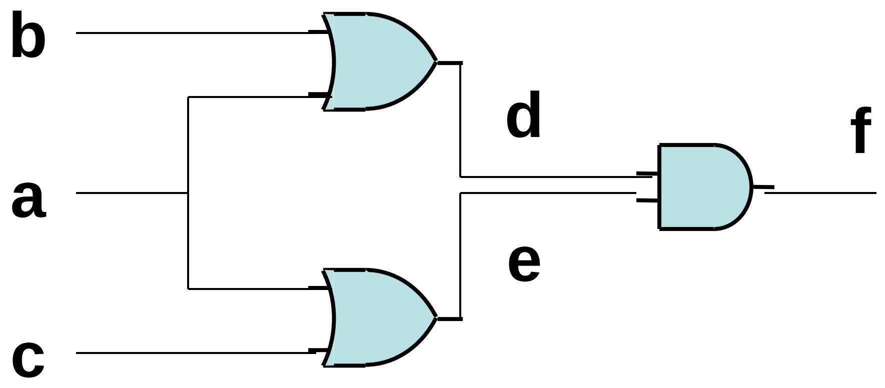
The Timeline



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

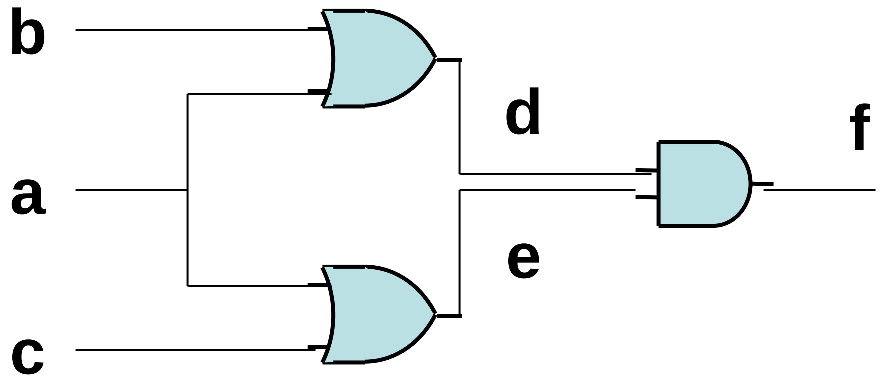
SOCRATES

M. H. Schulz, E. Auth, “SOCRATES: A highly efficient automatic test pattern generation system”, *IEEE Trans. Computers* C-37, 7:126-137, 1988



$a=1 \Rightarrow f=1$

SOCRATES

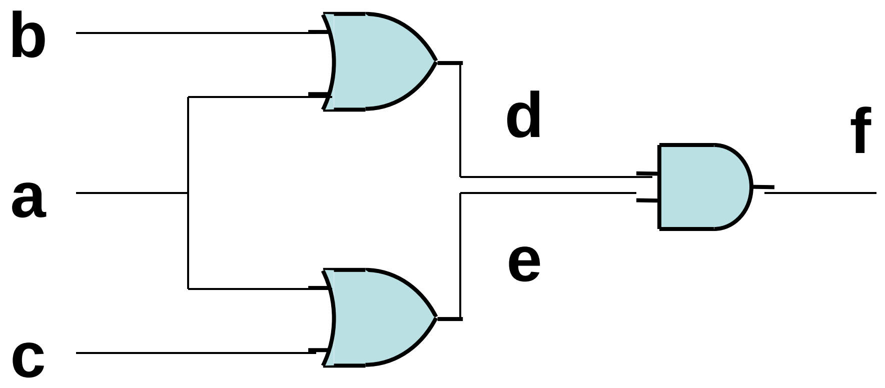


f=0⇒a=0

- Key idea : Use contra-positive for learning invariants
- Application to SAT : Add new clauses to the CNF, limits search space through faster backtrack

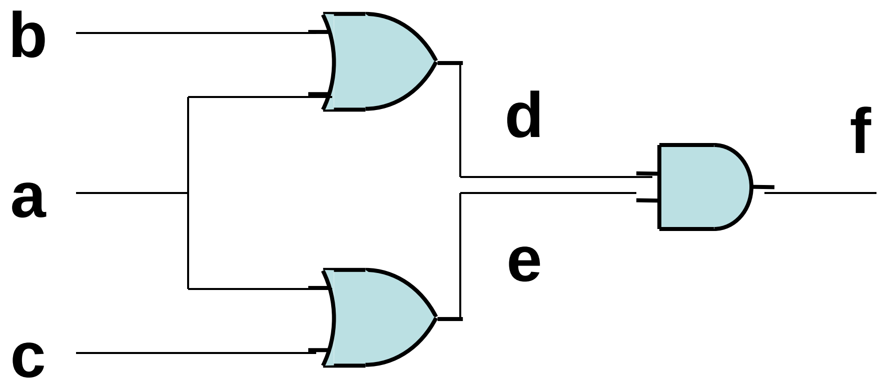
Hannibal

W. Kunz, “HANNIBAL: An efficient tool for logic verification based on recursive learning”, *Proc. ICCAD*, 1993



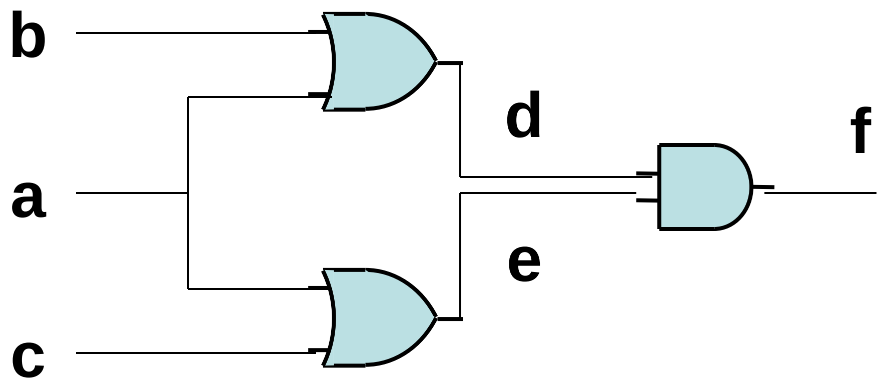
f=0⇒d=0 or e=0

Hannibal



Try d=0, Then a=0 and b=0

Hannibal

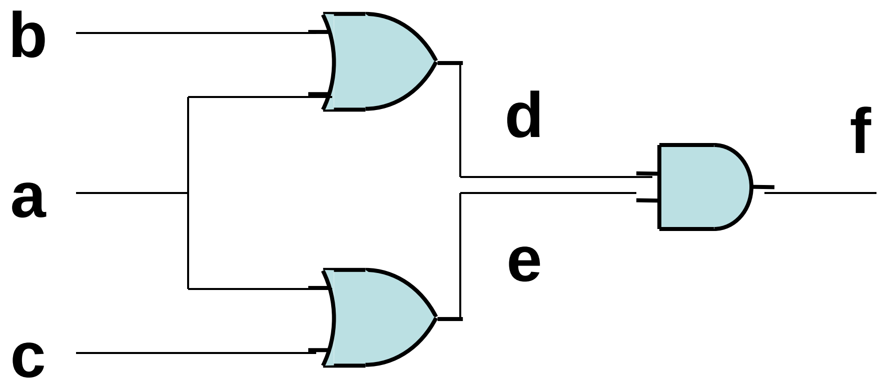


Try e=0, Then a=0 and c=0

Hannibal

Try d=0, Then a=0 and b=0

Try e=0, Then a=0 and c=0

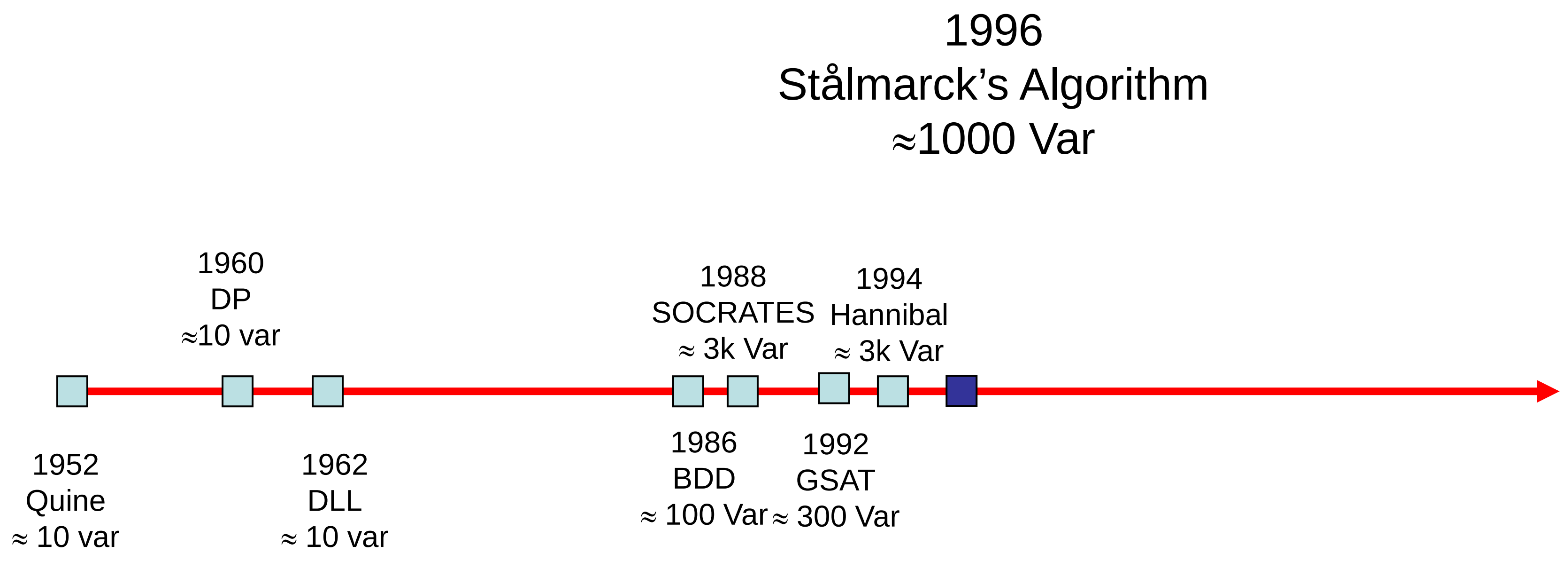


In both cases, a=0, so f=0 ⇒a=0

EDA Drivers

- Advances in ATPG and equivalence checking drove developments in SAT solvers
  - SOCRATES first incorporated learning
    - If P⇒Q, then Q’⇒P’
  - Hannibal used Recursive Learning with certain recursion depth
  - SOCRATES, Hannibal were able to handle practical sized circuits
- Use circuit specific information, so cannot immediately generalize this to arbitrary formulae

The Timeline



Gunnar M. N. Stalmarck, "System for determining propositional logic theorems by applying values and rules to triplets that are generated from Boolean formula", U. S. Patent 5276897, Issued 01/04/1994 (Filing Date: 06/14/1990) . European Patent EP0403454, Issued 06/28/1995 (Filing Date: 05/17/1990)

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

Stålmarck's algorithm

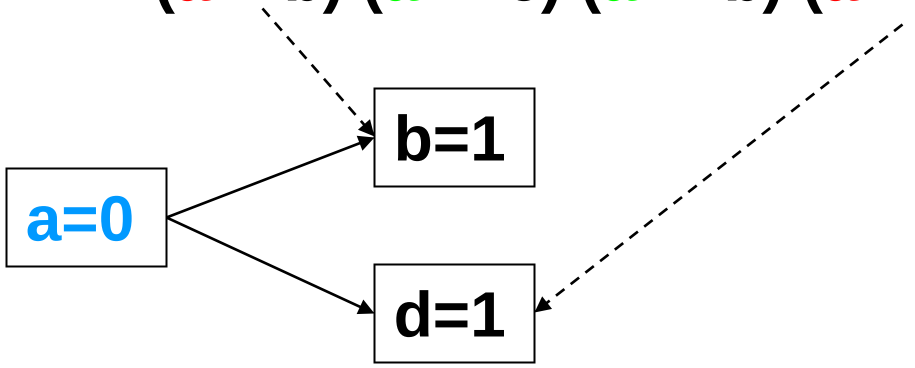
- Try both sides of a branch to find forced decisions (relationships between variables)

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

Stålmarck's algorithm

- Try both sides of a branch to find forced decisions

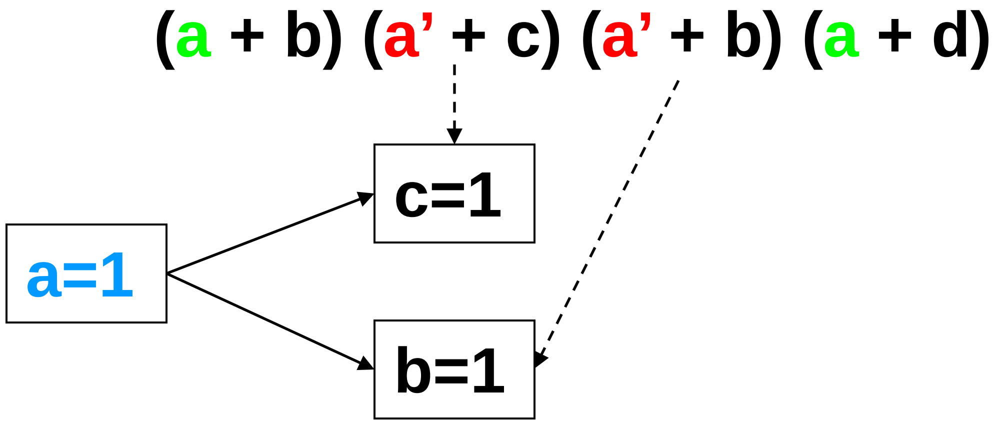
$$(a + b)\,(a' + c)\,(a' + b)\,(a + d)$$



$$\boxed{a=0 \Rightarrow b=1, d=1}$$

Stålmarck's algorithm

- Try both sides of a branch to find forced decisions



$\boxed{a=0 \Rightarrow b=1, d=1}$

$\boxed{a=1 \Rightarrow b=1, c=1}$

Stålmarck's algorithm

- Try both sides of a branch to find forced decisions

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

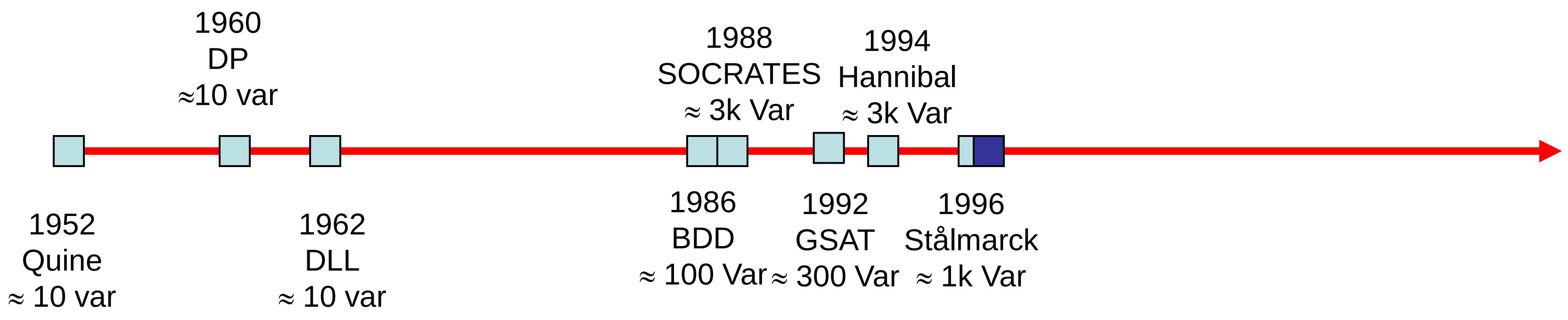
$$\boxed{a=0 \Rightarrow b=1, d=1} \quad \Rightarrow \; b=1$$

$$\boxed{a=1 \Rightarrow b=1, c=1}$$

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

The Timeline

1996
GRASP
Conflict Driven Learning,
Non-chornological Backtracking
≈1k Var



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.