# *A brief overview of SAT Solvers*

Sandro Spina

# Preliminaries – What is SAT?

- Propostional SATisfiability (SAT) addresses the problem of deciding whether a particular Boolean formula in some normal form has an assignment to its terms that makes the formula true.
- We know that SAT is NP-complete [Cook]
- In a way SAT provides a generic 'language' which can be used to express NP-complete problems in verification, scheduling, etc… A *SATsolver* is used to search for solutions to the problem.
- During this hour we shall be looking at a number of implementations of SAT solvers …

# SAT Solvers Input and Output

- The **input** to the SAT solver is a Boolean Propositional formula in some normal form (ex. Conjunctive Normal Form)
- Ex. $(a \lor \neg c) \land (\neg a) \land (b \lor c \lor \neg a)$
  - a,b and c are *variables*
  - A *literal* is either a +ve or a –ve instance of a variable
  - A CNF formula is a conjunction of one or more *clauses*, each of which is a disjunction of one or more literals
- The **output** of a SAT solver only needs to answer true (SAT) or false (UNSAT and therefore proved unsatisfiable) depending on the satisfiability of the formula. However, if the instance is SAT, most solvers can also output a satisfying assignment of the instance.

# Complete and Incomplete Algorithms

- **Incomplete methods** :- aim at finding solutions by heuristic means, without exhaustively exploring the space. Of course the problem is that these methods are not capable of detecting that *no solution* exists. Most incomplete methods are based on stochastic local (random) moves (ex. genetic algorithms, hill climbing techniques, GSAT). Clearly these SAT solvers are not very useful for verification.

- **Complete methods** :- aim at exploring the whole solution space, typically using some form of backtrack search. Resolution is one complete method. *Pruning* techniques are typically used to determine that certain regions contain no solution. (ex. DP, DLL, CHAFF, Stalmarck, etc)

# *David Putnam resolution based SATsolver (i)*

- **Rule 1**. Rule for the elimination of the one literal clause.
  - If F (in CNF) contains atomic formula p as a one-literal clause and also ¬p as a one-literal clause then F may be replaced by 0. $p \land \neg p$ = false
  - If an atomic formula p appears as a clause in F, then one many produce F′ such that F′ does not contain the clauses that contain p and deleting all occurrences of ¬p from the remaining clauses
  - If an atomic formula ¬p appears then obtain F′ by striking out all occurrences of p from the remaining clauses of F and deleting all one-literal clauses ¬p.
  - If F′ becomes empty after the application of these rules then the original formula F is SAT.

# *David Putnam resolution based SATsolver (ii)*

- **Rule 2**. Affirmative-Negative Rule.
  - If an atomic formula p occurs in a formula F only affirmatively or only negatively then all clauses which contain p may be deleted.
  - Ex. $(a \lor \neg c) \land (\neg a) \land (b \lor c \lor \neg a) \land b$
  - Can be reduced to $(a \lor \neg c) \land (\neg a)$

# *David Putnam resolution based SATsolver (iii)*

- **Rule 3**. Rule for eliminating atomic formulas
  - The following reduction is applied
  - $(A \vee p) \wedge (B \vee \neg p) \wedge R \Rightarrow (A \vee B) \wedge R$
  - Note that A,B and R are free of variable p
- The DP algorithm iteratively executes these three rules. If a contradiction results than the original formula is SAT.
- Note that usually the original formula to prove is negated since contradictions are usually produced faster.

# *Davis Logemann Loveland (DLL62) technique (i)*

- Note that resolution based methods (ex. DP) will not produce an assignment to the variables whenever the formula is SAT.
- DLL is a backtracking algorithm which in the case of SAT returns an assignment to the variables in the formula.
- DLL applies the same rules of DP except for the Rule 3.

# *Davis Logemann Loveland (DLL62) technique (ii)*

- Rule 3 (splitting rule).
  - Let $F = (A \lor p) \land (B \lor \neg p) \land R$
  - Then F is inconsistent if and only if
    - $A \land R$ and $B \land R$ are both inconsistent
  - The assignment of p to either T or F, splits the search space.
  - This is essentially a **depth first search**

# *Davis Logemann Loveland (DLL62) technique (example)*

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

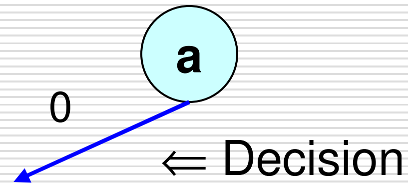
# Basic DLL Procedure - DFS

(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

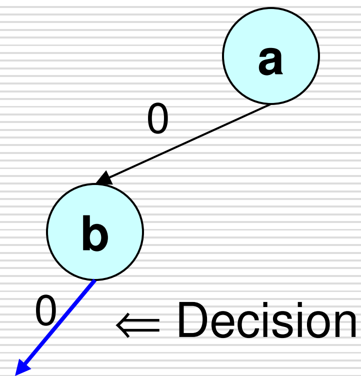
# Basic DLL Procedure - DFS

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



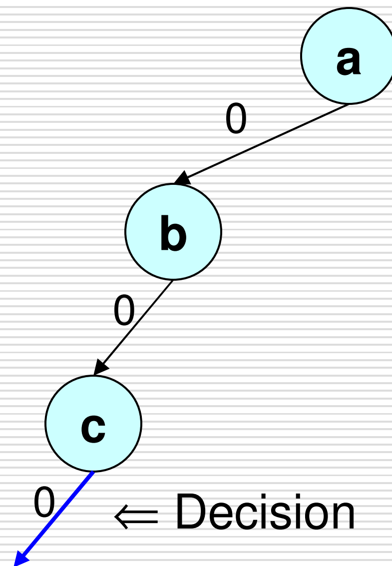
# Basic DLL Procedure - DFS

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



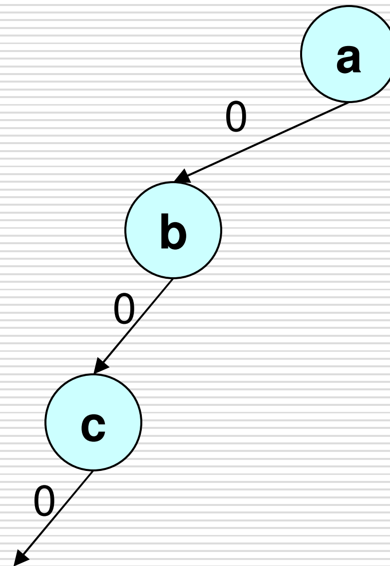
# Basic DLL Procedure - DFS

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

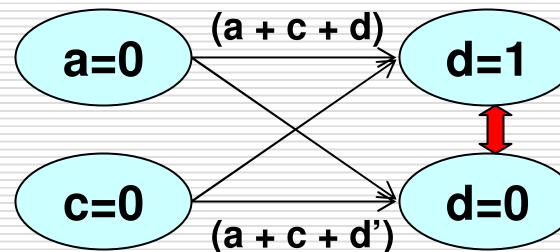


# Basic DLL Procedure - DFS

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



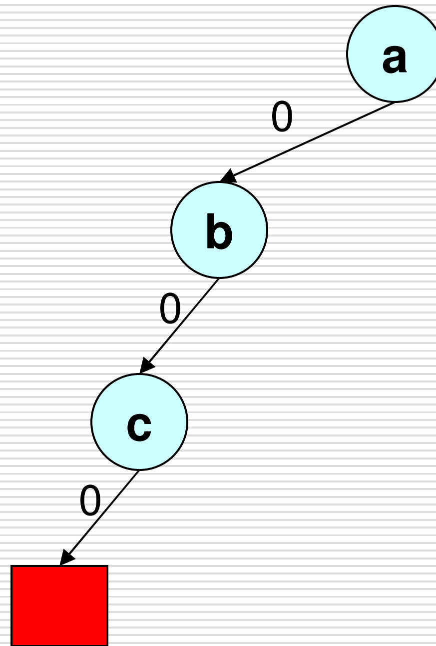

Implication Graph



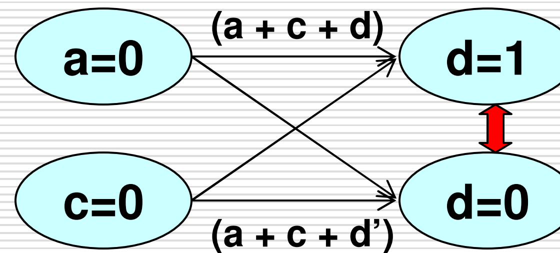

Conflict!

# 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)



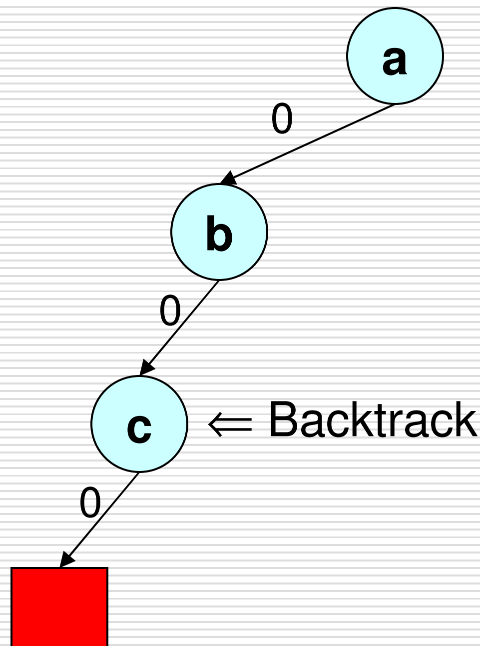

Implication Graph




Conflict!

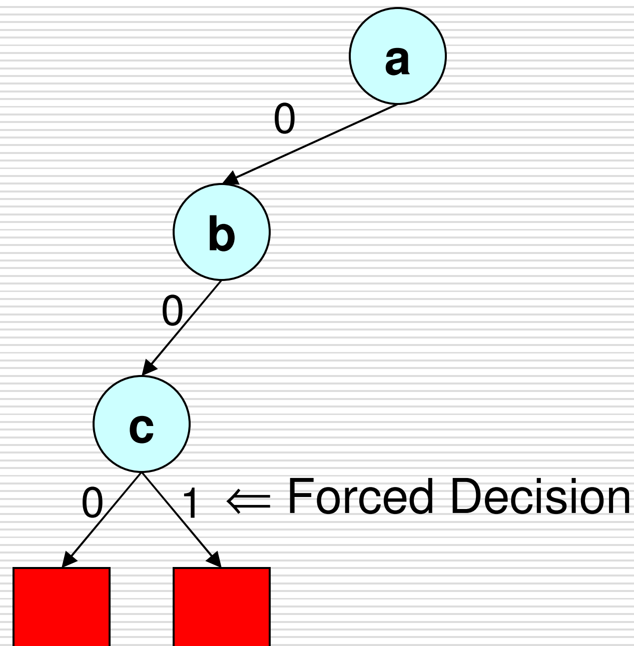
# Basic DLL Procedure - DFS

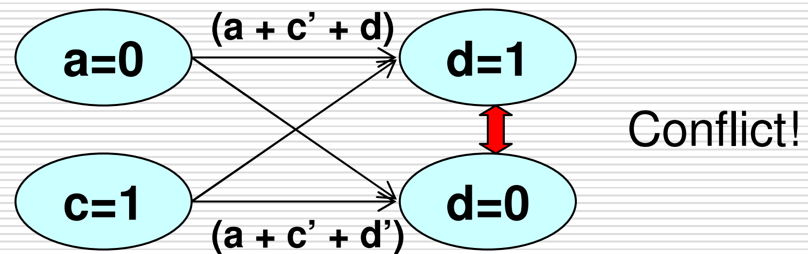
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



# Basic DLL Procedure - DFS

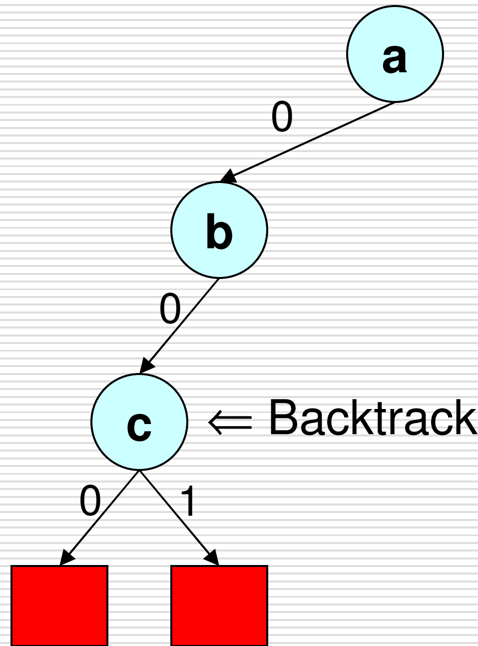
| |
|---|
| (a’ + b + c) |
| (a + c + d) |
| (a + c + d’) |
| (a + c’ + d) |
| (a + c’ + d’) |
| (b’ + c’ + d) |
| (a’ + b + c’) |
| (a’ + b’ + c) |



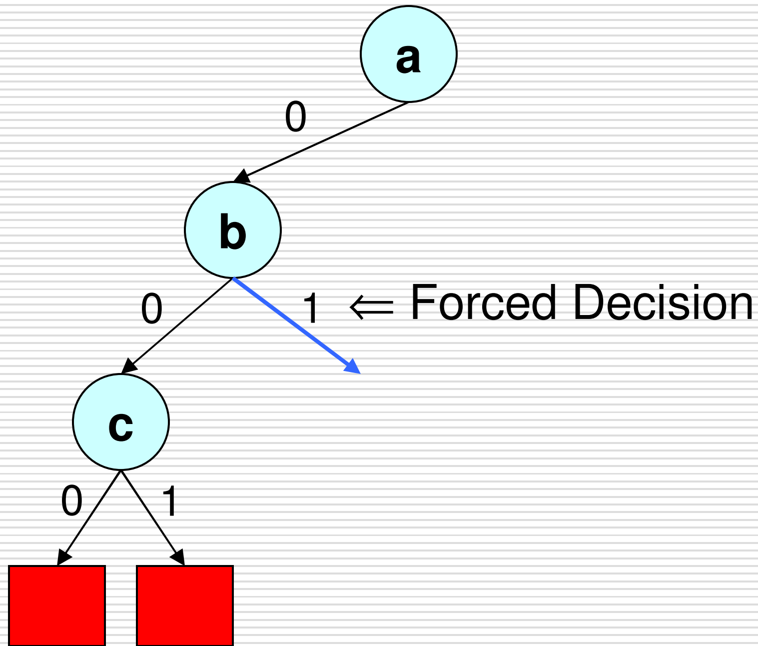
# Basic DLL Procedure - DFS

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



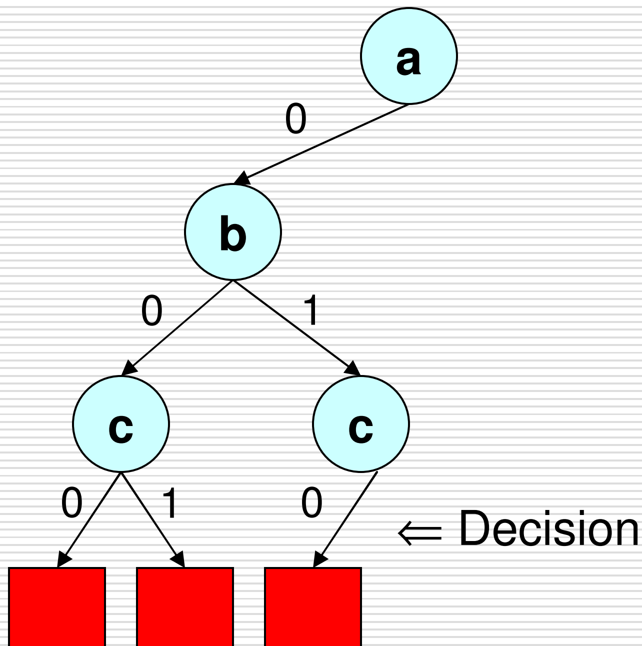
# Basic DLL Procedure - DFS

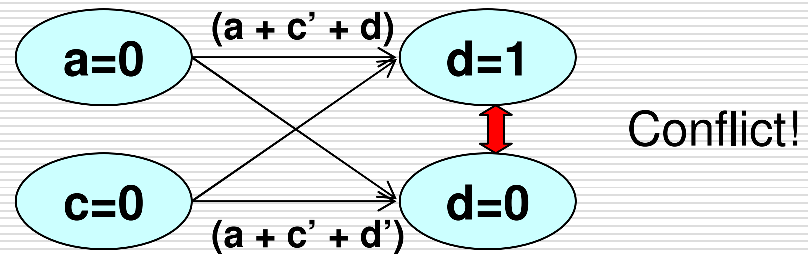
**(a’ + b + c)**
**(a + c + d)**
**(a + c + d’)**
**(a + c’ + d)**
**(a + c’ + d’)**
**(b’ + c’ + d)**
**(a’ + b + c’)**
**(a’ + b’ + c)**



# Basic DLL Procedure - DFS

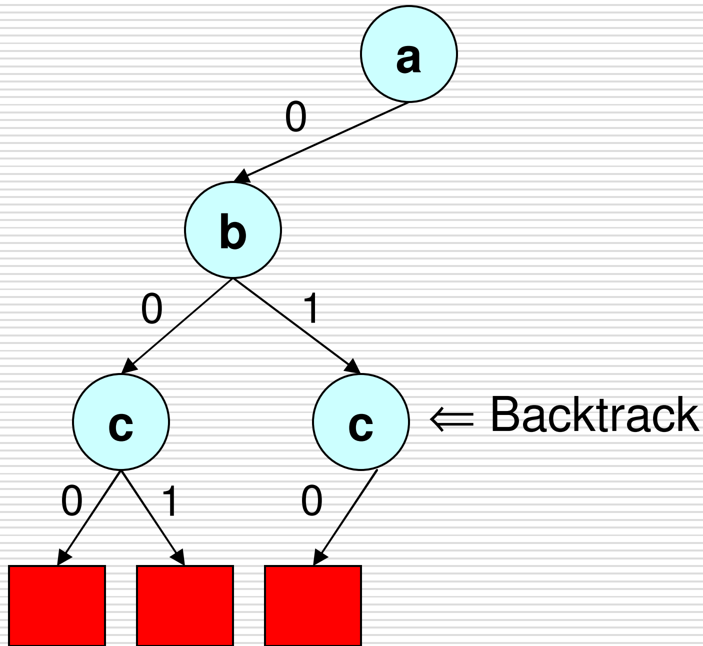
| (a' + b + c) |
|---|
| (a + c + d) |
| (a + c + d') |
| (a + c' + d) |
| (a + c' + d') |
| (b' + c' + d) |
| (a' + b + c') |
| (a' + b' + c) |



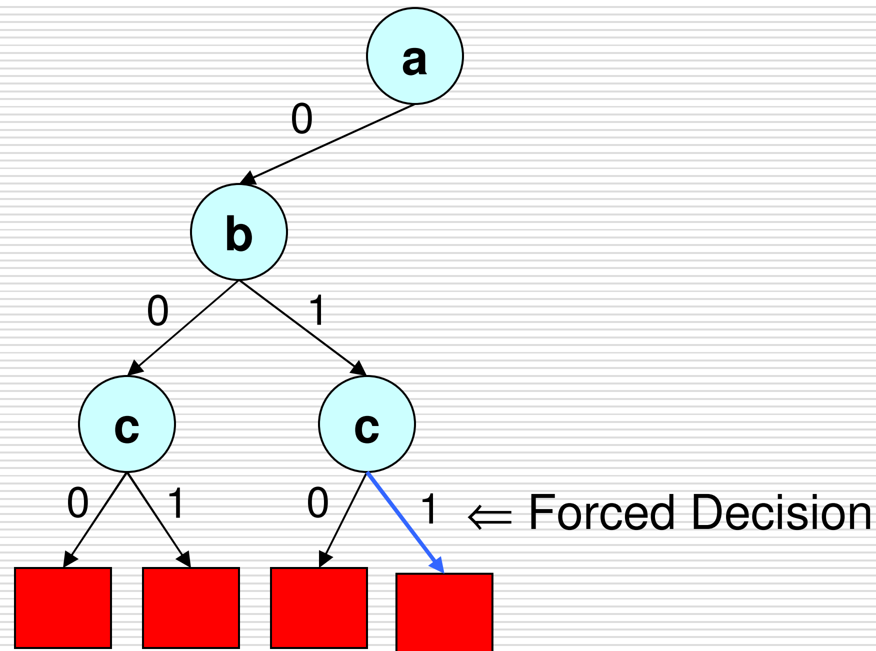
# Basic DLL Procedure - DFS

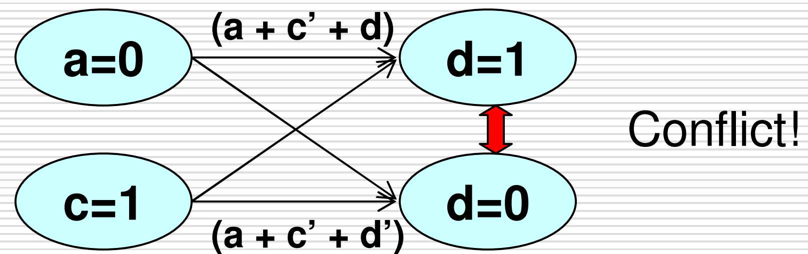
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



# Basic DLL Procedure - DFS

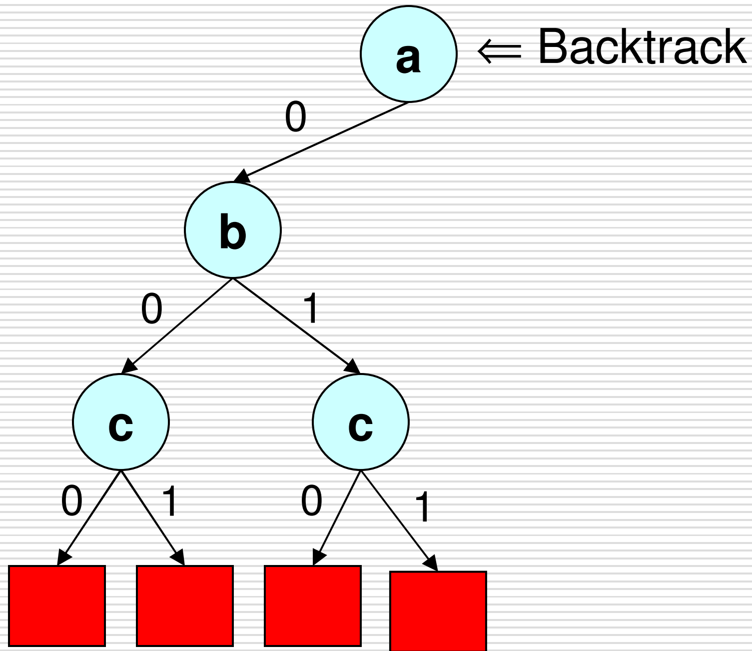
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



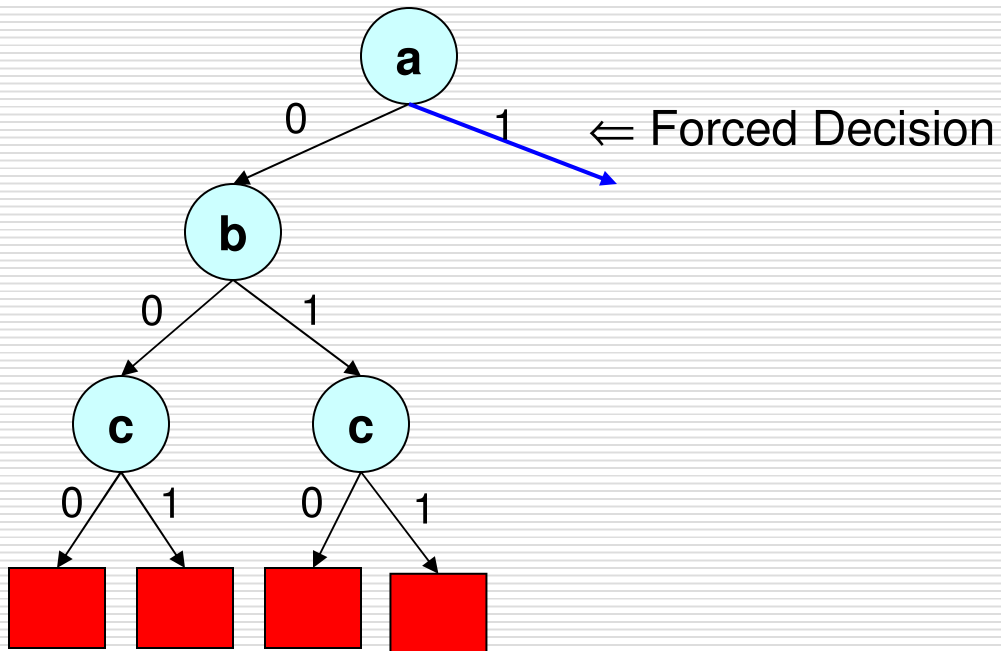
# Basic DLL Procedure - DFS

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



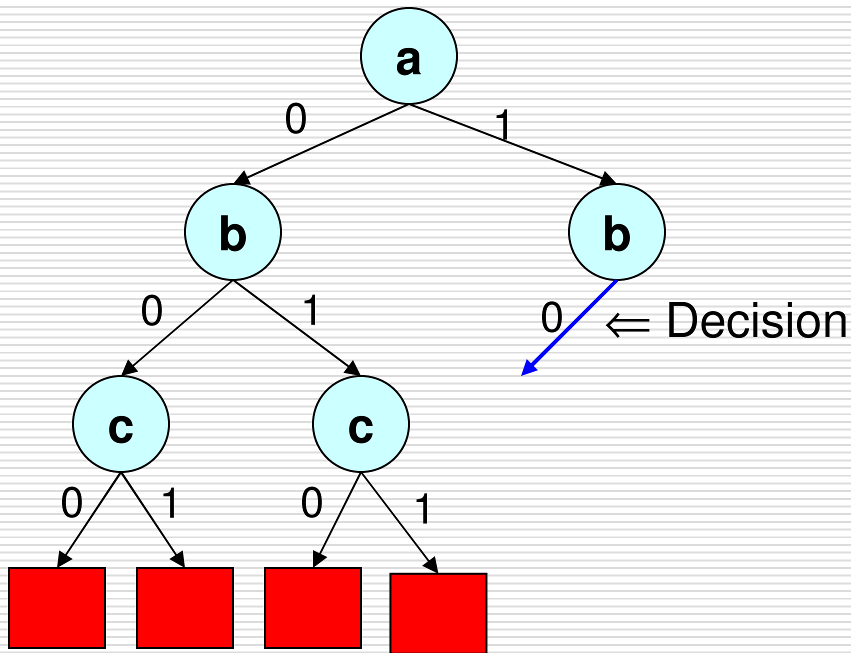
# Basic DLL Procedure - DFS

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



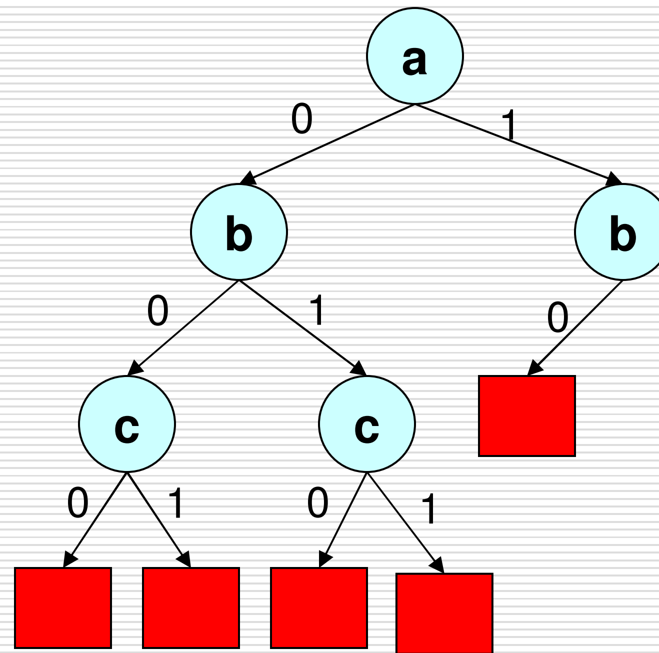
# Basic DLL Procedure - DFS

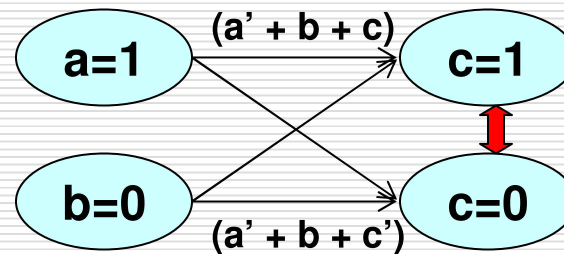
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



# 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)

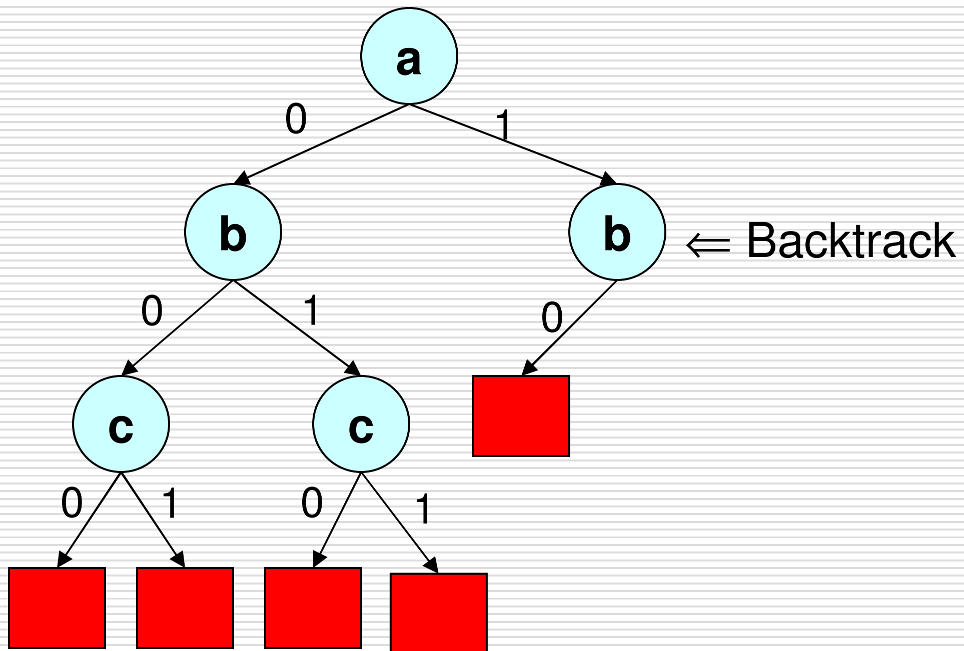



Conflict!

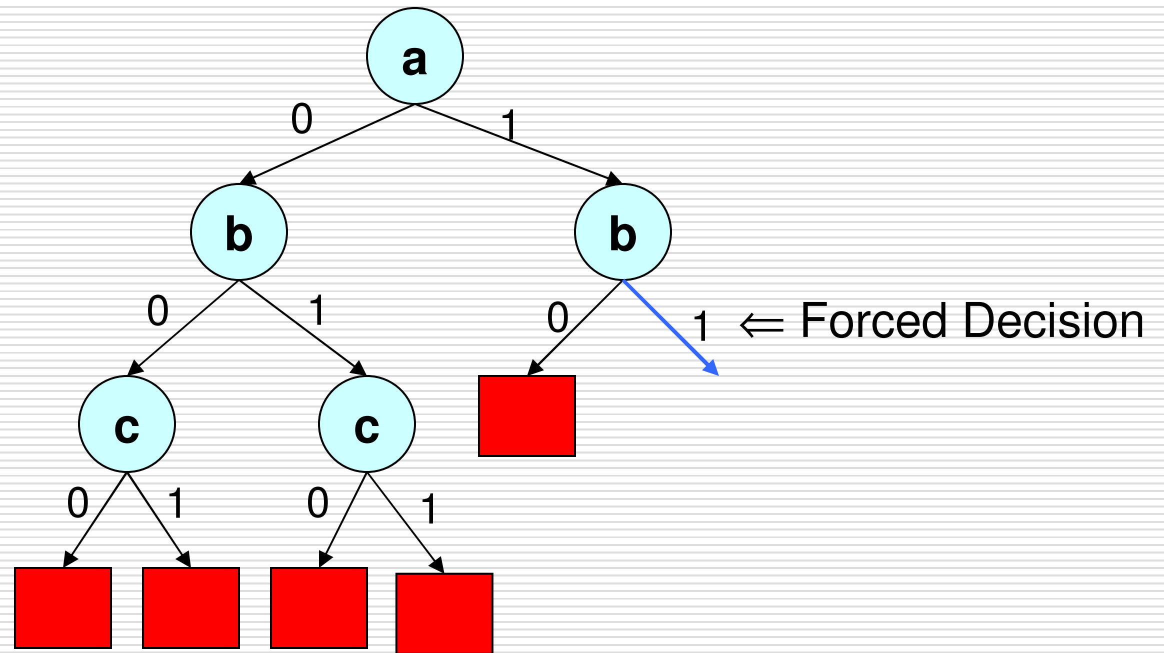
# Basic DLL Procedure - DFS

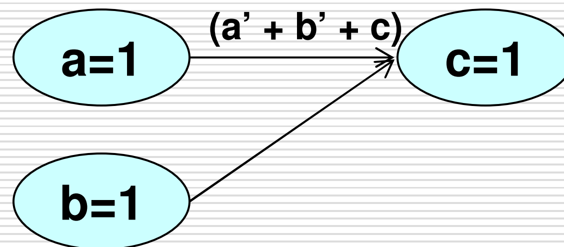
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



# Basic DLL Procedure - DFS

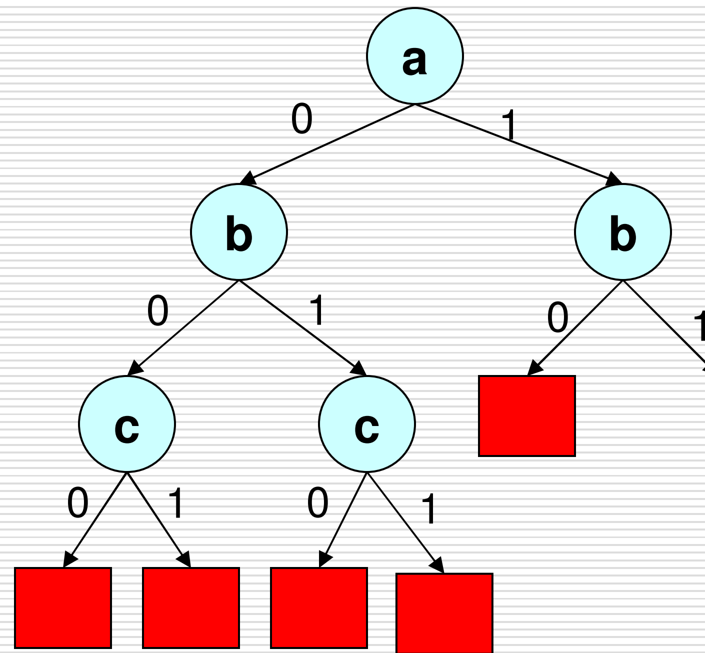
**(a' + b + c)**
**(a + c + d)**
**(a + c + d')**
**(a + c' + d)**
**(a + c' + d')**
**(b' + c' + d)**
**(a' + b + c')**
**(a' + b' + c)**

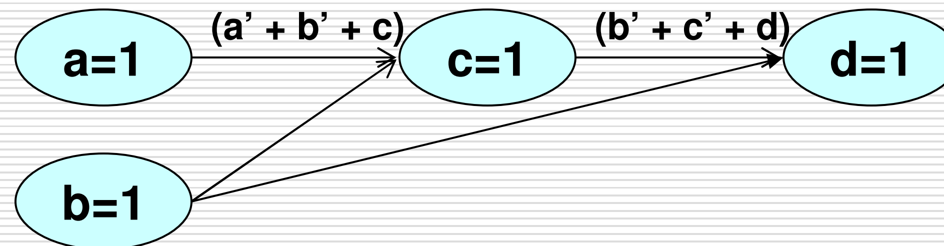


# Basic DLL Procedure - DFS

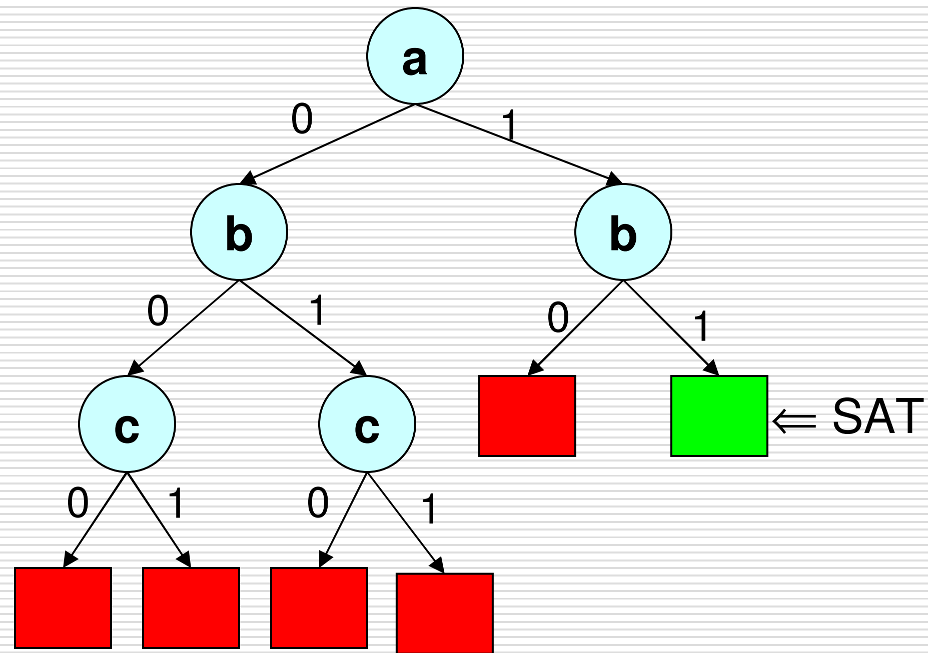
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

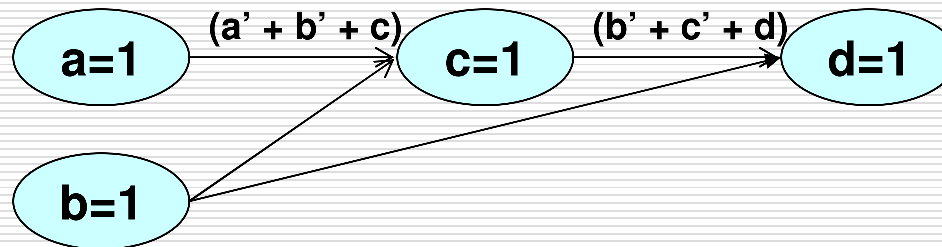


# Basic DLL Procedure - DFS

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



# Stalmarck's Method for propositional logic

- The normal form used in this method makes use only of implication and false.
  - $A \vee B \quad = \quad \sim A \rightarrow B$
  - $A \wedge B \quad = \quad \sim(A \rightarrow \sim B)$
  - $\sim\sim A \quad = \quad A$
  - $\sim A \quad = \quad A \rightarrow False$
- The formula $x \leftrightarrow (y \rightarrow z)$ is represented by the triple (x, y, z)
  - For example $p \rightarrow ( q \rightarrow p)$ becomes
  - (b1, q, p)
  - (b2, p, b1)
- In order to prove a formula valid, first assume it to be false then try to derive a contradiction.

## *Stalmarck's Method for propositional logic (ii)*

- Simple rules. A simple rule takes a triggering triplet and derives new information about its variables.
- For example we know that if y → z is false, then y must be true and z false. This information is valid throughout the whole derivation.
- The crucial point to note here is that this method is performing a *breadth first search* within the search space.

## *Stalmarck's Method for propositional logic (example)*

- Ex. P → ( q → p)
  - (b1, q, p)
  - (0, p, b1) →
  - (b1, q, p)[p/1, b1/0] →
  - (0,q,1) (a contradictory/terminal triplet !! Halt and SAT)
- The other terminal triplets are :
  - (1, 1, 0) and (0, 0, x)
- There are 6 different simple rules

# Stalmarck's Method for propositional logic (dilemma)

- Dilemma (to be or not to be) Rule.

$$\frac{\begin{array}{cc} & T \\ \hline T[x/1] & T[x/0] \\ D_1 & D_2 \\ U[S_1] & V[S_2] \end{array}}{T[S]}$$

- If one of these derivations gives a terminal triplet, then the result of applying the rule is the result of the other derivation.

- !!! Very important !!! If neither $D_1$ nor $D_2$ leads to a contradiction, then the resulting substitution is the intersection of $S_1$ and $S_2$. That is, any information gained both from assuming that *x* is true and from assuming that *x* is false must hold independent of the value of *x*.

# *SAT heuristics*

- The main research focus on SAT branching heuristics is to *discover conflicts as early as possible.*
- A heuristic should be also cheap to evaluate.
- Examples include (depends on SAT test samples)
  - Maximum Occurrences on Minimum sized clauses (MOM) – try to produce large numbers of implications or to satisfy as many clauses as possible.
  - *Dynamic Largest Individual Sum* (DLIS) – selects the literal that appears most frequently in unresolved clauses.
  - Variable State Independent Decaying Sum (VSIDS) – used in the CHAFF SAT solver.

# *Variable State Independent Decaying Sum (VSIDS) - CHAFF Heuristic for decision()*

1. Each variable in each polarity has a counter, initialized to 0.
2. When a clause is added to the database, the counter associated with each literal in the clause is incremented
3. The (unassigned) variable and polarity with the highest counter is chosen at each decision
4. Ties are broken randomly by default, although this is configurable
5. Periodically, all the counters are divided by a constant

- In CHAFF a clause learning mechanism (from conflicts) adds new clauses (and literals) to the clause database as the search progresses. VSIDS score is a literal occurrence count with higher weight on the more recently added literals.

# *Conclusions*

- The aim of this talk was to describe a number of SAT solvers.
- Hopefully you got a better idea of what SAT is all about…
- Thanks for attending this SVRG talk!