COS 326: Functional Programming

Lecture

Boolean Satisfiability (SAT) Solvers

Aarti Gupta

Acknowledgements: Sharad Malik, Emina Torlak

What is SAT?

Given a propositional logic (Boolean) formula,

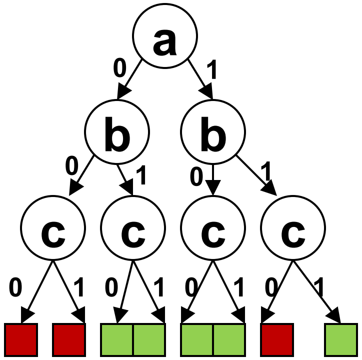
find a variable assignment such that the formula evaluates to true,

or prove that no such assignment exists.

F = (a || b) && (a’ || b’ || c)

|| denotes OR
&& denotes AND
’ denotes NOT

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

SAT and logics

Propositional logic is a subset of

- First order logic
- Higher-order logic

Validity of formulas (i.e., checking if *all* variable assignments make the formula true)

- Propositional logic: decidable
- First order logic: semi-decidable
- Arithmetic and higher-order logic: undecidable

Complexity of SAT: NP-complete

- There is no known polynomial-time algorithm
- … but often tractable in practice on real-world problems!

Why is it of interest *now*?

SAT: is a Boolean formula f satisfiable?

1988
SOCRATES
≈ 3k Var

1996
GRASP
≈1k Var

2001
Chaff
≈10k var

1952
Quine
≈ 10 var

1960
DP
≈10 var

1962
DLL
≈ 10 var

1986
BDD
≈ 100 Var

1996
Stålmarck
≈ 1k Var

1997
SATO
≈ 1k Var

2003
MiniSAT
≈10k var

(Source: Sharad Malik)

SMT (Satisfiability Modulo Theory):

is a first-order logic formula theory-satisfiable?

2008
Z3

2004
DPLL(T)

2009
SM

Where are we today?

Intractability of the problem no longer daunting

- can regularly solve practical instances with ***millions*** of variables and constraints

SAT has matured from theoretical interest to practical impact

- Electronic Design Automation (EDA)
  - Widely used in many aspects of chip design: equivalence checking, assertion verification, synthesis, debugging, post-silicon validation
- Software verification
  - Commercial use at Microsoft, Amazon, Google, Facebook, …
- AI and Planning problems

- CAV 2009 Award due to industrial impact
  - Chaff solver team from Princeton shared the award!

Where are we today?

Significant SAT community

- SatLive Portal (http://www.satlive.org/)
- Annual SAT competitions (http://www.satcompetition.org/)
- SAT Conference (http://www.satisfiability.org/)

Emboldened researchers to take on even harder problems related to SAT

- Max-SAT: for optimization
- Satisfiability Modulo Theories (SMT): for more expressive theories
- Quantified Boolean Formulas (QBF): for more complex problems

- Many ideas from SAT solvers are applied here

Some basics first …

Boolean formulas: Syntax

Formula f =	// inductive definition
true | false |	// base cases: constants
v |	// base case: variable
NOT g | g AND h | g OR h	// inductive cases

where

v is a *propositional* variable, i.e., it takes value true or false

NOT, AND, OR are the usual Boolean operators

Boolean formulas: Semantics

Given a Boolean formula f,

and an *Interpretation M*, which maps variables to true/false

We can *evaluate* f under M to produce a Boolean result (true or false).

- Base case true: return true
- Base case false: return false
- Base case variable v: return value of v in M
- Inductive cases: return result by using the *truth tables* shown below

g	**Not g**
0	1
1	0

g	**h**	**g AND h**
0	0	0
0	1	0
1	0	0
1	1	1

g	**h**	**g OR h**
0	0	0
0	1	1
1	0	1
1	1	1

0 denotes false
1 denotes true

Example: Evaluate f: (a OR b) AND (NOT c) under M:{a ↦ true, b ↦ false, c ↦ false}

f = (1 OR 0) AND (NOT 0) *… f evaluates to true under M*

Boolean formulas: Semantics

Given a Boolean formula f,

and an Interpretation M, which maps variables to true/false

If f evaluates to true under M, we say that *M satisfies f*

|| denotes OR
&& denotes AND
’ denotes NOT

Example: f: (a || b) && (a’ || b’ || c), M1: {a ↦ true, b ↦ true, c ↦ false}

(Q1) Does M1 satisfy f?

No, because f evaluates to false under M1.

(Q2) Is f satisfiable, i.e., does there exist an M such that M satisfies f?

Yes, f is satisfiable.

For example, M2: {a ↦ true, b ↦ true, c ↦ true} satisfies f

SAT solvers can automatically find a satisfying interpretation! (if it exists)

SAT solvers: a condensed history

Deductive

- Davis-Putnam 1960 [DP]
- Based on “resolution”

Backtracking Search

- Davis, Putnam, Logemann and Loveland 1962 [DLL, DPLL]
- Exhaustive search for satisfying assignment

Conflict Driven Clause Learning [CDCL]

- GRASP: Integrate a constraint learning procedure, 1996

Locality Based Search

- Emphasis on exhausting local sub-spaces, e.g., Chaff, 2001
- Added focus on efficient implementations

Princeton Senior Thesis!

“Pre-processing”

- Peephole optimization, e.g., MiniSat, 2005

…

SAT problem representation

Boolean formulas represented in:
Conjunctive Normal Form (CNF)

(a || b || c) && (a’ || b’|| c) && (a’|| b || c’) && (a || b’ || c’)

formula: is a conjunction of clauses

clause: is a disjunction of literals e.g., (a || b’ || c’)

literal: is a variable or its negation e.g., a, a’

- for formula to be true: ***each*** clause must be satisfied
- in each clause: ***some*** literal must be true

CNF Representation

Q: Can any Boolean formula be converted to CNF?

Yes!

Q: How can I convert a Boolean formula to CNF?

Many different translations exist …

You will explore two translations in Assignment 3

1. A naïve translation based on de Morgan’s Laws
2. Tseitin transformation (useful for checking SAT)

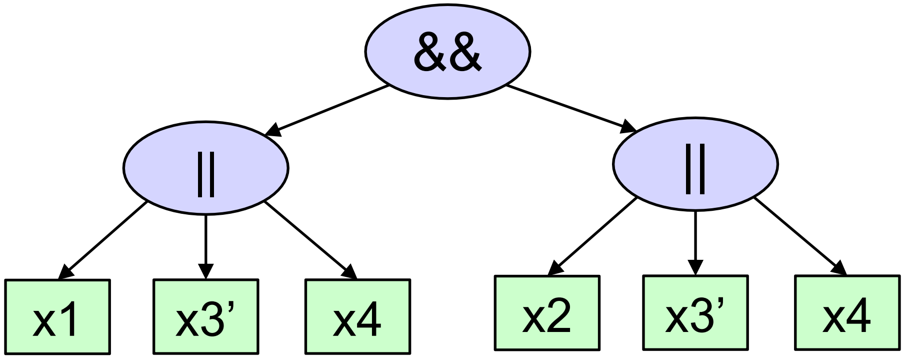
Translation to CNF

We can view CNF as a special kind of expression tree for a Boolean formula

CNF has maximum 3 levels, where

- top-level node is AND
- second-level nodes are OR
- NOT (i.e., negation) can be optionally applied *only at the leaves*
- leaf level nodes are variables (with/without negation)

Example: (x1 || x3’ || x4) && (x2 || x3’ || x4) is in CNF



|| denotes OR
&& denotes AND
’ denotes NOT

(1) Naïve Translation to CNF

Given an *arbitrary* expression tree for a Boolean formula

- #1: Push NOT nodes inside an AND or OR

  de Morgan's Laws
  NOT (p || q) = (NOT p) && (NOT q)
  NOT (p && q) = (NOT p) || (NOT q)

- #2: Distribute outer OR nodes over an inner AND

  (p && q) || (x && y) = (p || x) && (p || y) && (q || x) && (q || y)

- Simplifications
  - NOT(NOT p) = p
  - Nested AND nodes to a single AND (when possible)
  - Nested OR nodes to a single OR node (when possible)

Correctness of Naïve translation

- The generated CNF formula is *equivalent* to the given formula
- However, its size can be *exponentially bigger* ☹

(1) Naïve translation to CNF: Example

Example: (x1 && x2) || (NOT (x3 && NOT x4))

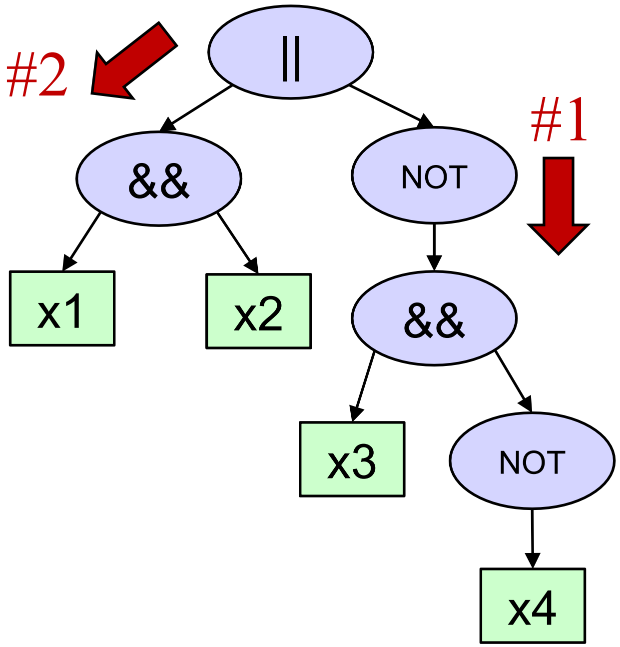
= (x1 && x2) || (NOT x3 || NOT(NOT x4)) … #1

= (x1 && x2) || (NOT x3 || x4) … NOT simplification

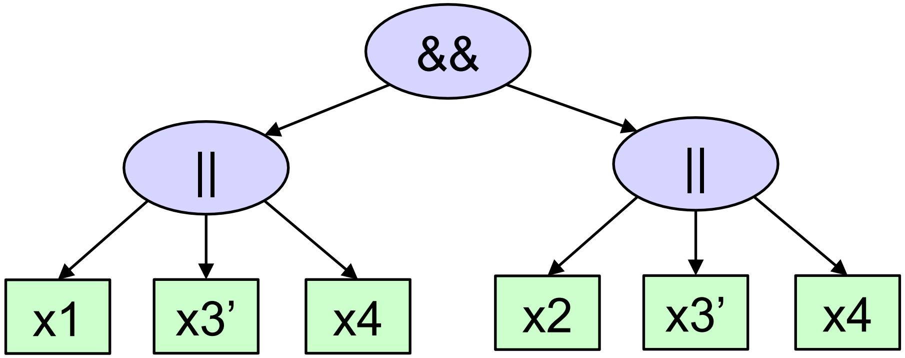
= (x1 || NOT x3 || x4) && (x2 || NOT x3 || x4) … #2

= (x1 || x3’ || x4) && (x2 || x3’ || x4)

Original formula

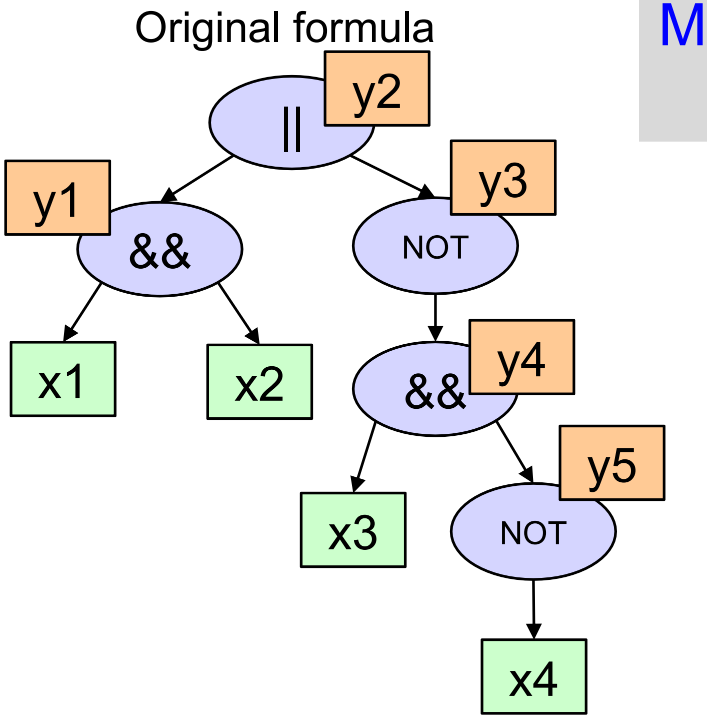


CNF representation



Tseitin Transformation Example

Main idea: Introduce fresh variable for each subformula and write ”equations”


Original formula

New variables: y1, y2, y3, y4, y5

Equations

y1 = x1 && x2
y2 = y1 || y3
y3 = NOT y4
y4 = x3 && y5
y5 = NOT x4

Equation	CNF to implement the Equation
z = NOT x	(x || z) && (x’ || z’)
z = x && y	(x || z’) && (y || z’) && (x’ || y’ || z)
z = x || y	(x’ || z) && (y’ || z) && (x || y || z’)

CNF

(x1 || y1’) && (x2 || y1’) && (x1’ || x2’ || y1) &&
(y1’ || y2) && (y3’ || y2) && (y1 || y3 || y2’) &&
(y3 || y4) && (y3’ || y4’) &&
(x3 || y4’) && (y5 || y4’) && (x3’ || y5’ || y4) &&
(x4 || y5) && (x4’ || y5’) &&
(y2)

(2) Tseitin Transformation

Main idea: Introduce new (fresh) variables for each subformula

- Write ”equations”: new variable = subformula
- Generate CNF for each equation, depending on the operator, as follows:

Equation	CNF to implement the Equation
z = NOT x	(x || z) && (x’ || z’)
z = x && y	(x || z’) && (y || z’) && (x’ || y’ || z)
z = x || y	(x’ || z) && (y’ || z) && (x || y || z’)

- AND together the CNFs for all equations
- AND a clause with a single literal for the top-level formula (if you want to check satisfiability of the top-level formula)

Correctness of Tseitin transformation

- For a given formula f, let Tseitin(f) denote the generated CNF formula
- Size of Tseitin(f) is *linear* in the size of f
- Tseitin(f) is *equi-satisfiable* with f
  - i.e., Tseitin(f) is satisfiable *if and only if* f is satisfiable

Boolean circuit to CNF

Tseitin transformation for Boolean (combinational) circuits

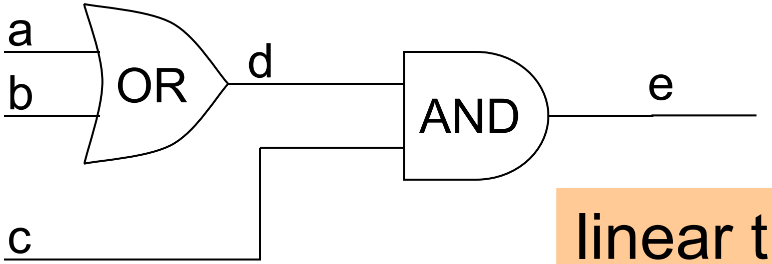
d ≡ (a || b)

(a || b || d')
(a' || d)
(b' || d)

e ≡ (c && d)

(c' || d' || e)
(d || e')
(c || e')

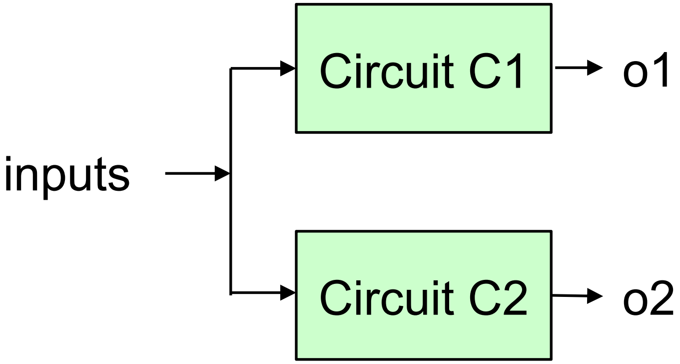
Tseitin rules



linear time conversion of any Boolean circuit into CNF using auxiliary variables

Applications of SAT

- Checking circuit equivalence



Given the same inputs,
are outputs o1 and o2 equivalent?

Formula f:

$$CNF(C1)\ \&\&\ CNF(C2)\ \&\&\ \overbrace{(o1\ ||\ o2)\ \&\&\ (o1'\ ||\ o2')}^{\text{o1 is NOT equivalent to o2}}$$

Use a SAT solver to check if formula f is satisfiable

- If f is satisfiable, then C1 and C2 are *not equivalent*
- If f is unsatisfiable, then C1 and C2 are *equivalent*

- *SAT solver checks over all inputs (without enumeration)!*

Another application

Automatic test generation

```
if (c1) then {
   if (c2) then {
   …
   else if (c3) then {
      …
      assert(c4)
      …
   }
}}
```

- Suppose you are testing a program to check if the assert can fail
- Note that the assert can fail only when c1 is true, c2 is false, c3 is true, and c4 is false.
- It may be difficult to *manually* come up with such an input

SAT solvers are used for automatically generating test inputs!

- Represent *int* variables in programs as 64-bit bitvectors
- Construct formula for checking satisfiability of *path-condition*
- Use SMT solvers, e.g., Z3 (which uses SAT solver for bitvectors)

SAT solvers: a condensed history

Deductive

- Davis-Putnam 1960 [DP]
- Based on “resolution”

➢ Backtracking Search

- Davis, Putnam, Logemann and Loveland 1962 [DLL, DPLL]
- Exhaustive search for satisfying assignment

Conflict Driven Clause Learning [CDCL]

- GRASP: Integrate a constraint learning procedure, 1996

Locality Based Search

- Emphasis on exhausting local sub-spaces, e.g., Chaff, 2001
- Added focus on efficient implementations

“Pre-processing”

- Peephole optimization, e.g., MiniSat, 2005

…

Basic DPLL Search

M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM, 5:394–397, 1962*

Performs backtracking search over variable assignments

- We will represent the formula in CNF
    - Implicitly a set of clauses (ie, the AND is implicit)
- We will make “decisions” by assigning values to variables
- We will keep track of a “decision tree” that records the current *partial assignment* to variables
- We will “backtrack” when the latest decision cannot lead to a satisfying assignment (“solution”)

During the search:

- If all clauses are satisfied, we have found a satisfying assignment (and can terminate)
- If we have exhausted all possible assignments without finding a solution, then the formula is unsatisfiable

Basic DPLL Search

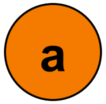
Definitions: under a given partial assignment (PA)

- A variable may be
  - **assigned** (true or false literal)
  - **unassigned**

- A clause may be
  - **satisfied** (≥1 true literal)
  - **unsatisfied** (all false literals)
  - **unit** (one unassigned literal, rest false)
  - **unresolved** (otherwise)

Basic DPLL Search

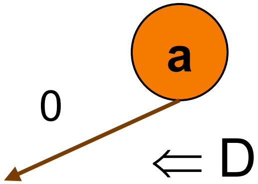
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



\+ denotes OR
’ denotes NOT

Basic DPLL Search

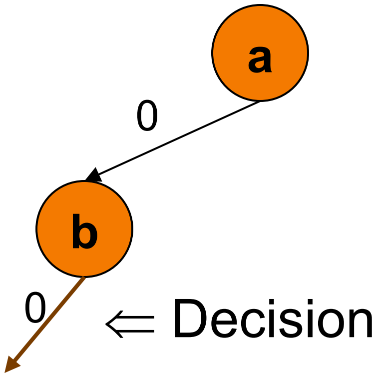
→ (a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
→ (a’ + b + c’)
→ (a’ + b’ + c)



⇐ Decision

Basic DPLL Search

(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 DPLL Search

(a’ + b + c)

(a + c + d)

(a + c + d’)

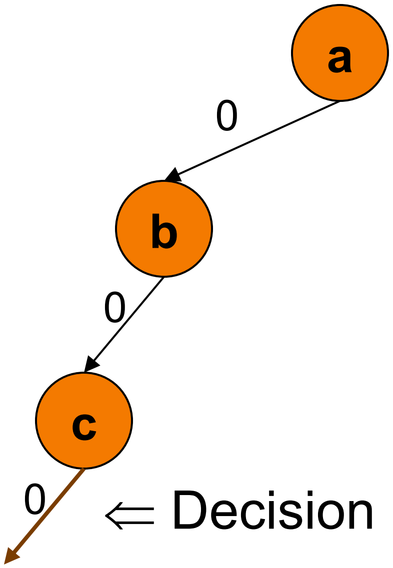
→ (a + c’ + d)

→ (a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

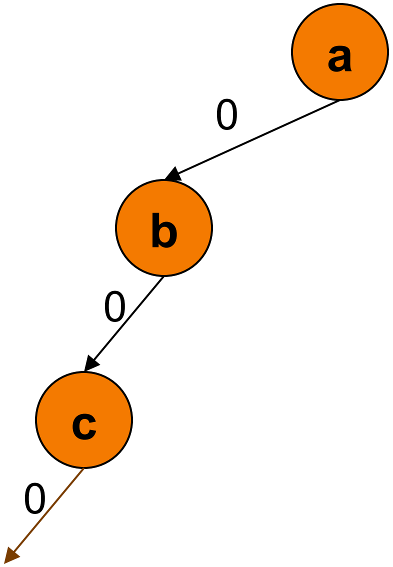
(a’ + b’ + c)



⇐ Decision

Basic DPLL Search

(a’ + b + c)
→ (a + c + d) ⇐ Unit
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



BCP: Boolean Constraint Propagation repeatedly applies *Unit Clause Rule*

If all but one literals in a clause are false, then the remaining literal is *implied* to true.

Basic DPLL Search

(a’ + b + c)

→ (a + c + d) ⇐ Unit

(a + c + d’)

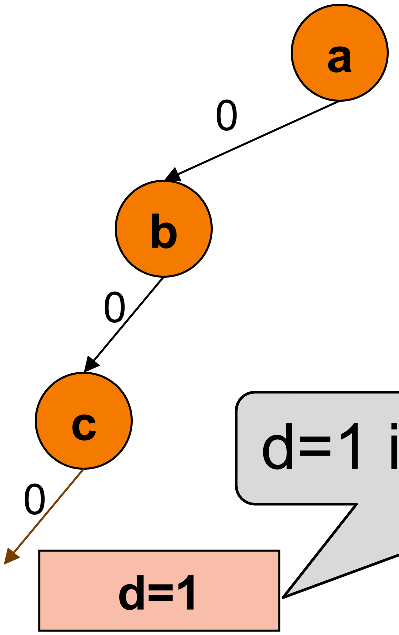
(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



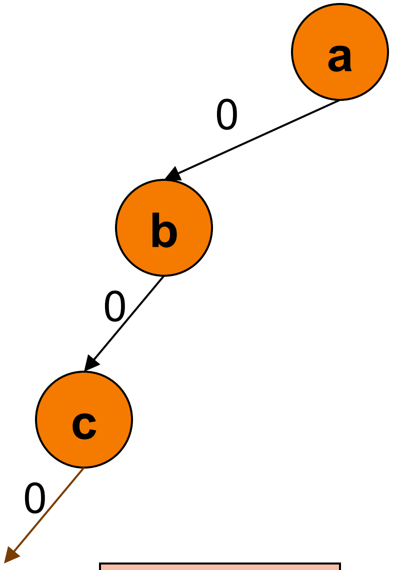
d=1 is implied at level 3

BCP: Boolean Constraint Propagation
repeatedly applies *Unit Clause Rule*

If all but one literals in a clause are false,
then the remaining literal is *implied* to true.

Basic DPLL Search

(a’ + b + c)
→ (a + c + d)
→ (a + c + d’) ⇐ Unit
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



d=1,d=0

Basic DPLL Search

(a’ + b + c)

→ (a + c + d)

→ (a + c + d’)

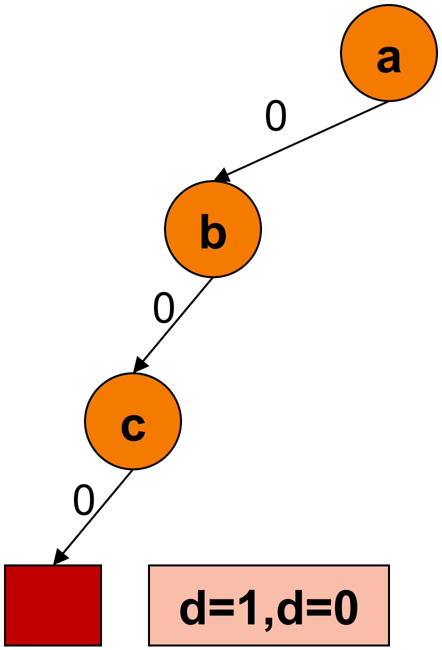
(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Conflict!

Basic DPLL Search

(a’ + b + c)

→ (a + c + d)

→ (a + c + d’)

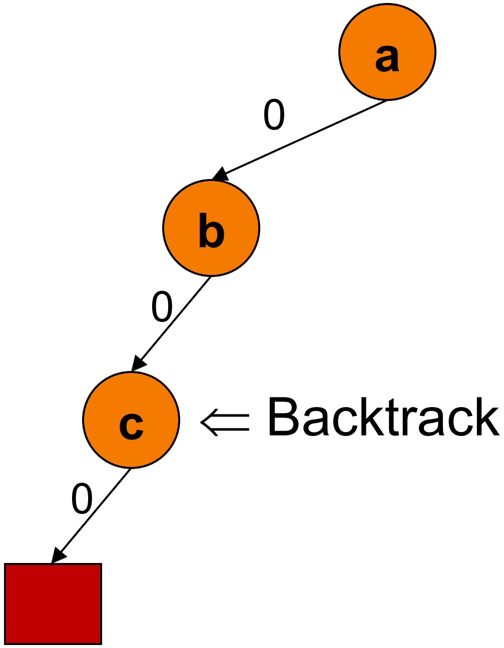
→ (a + c’ + d)

→ (a + c’ + d’)

(b’ + c’ + d)

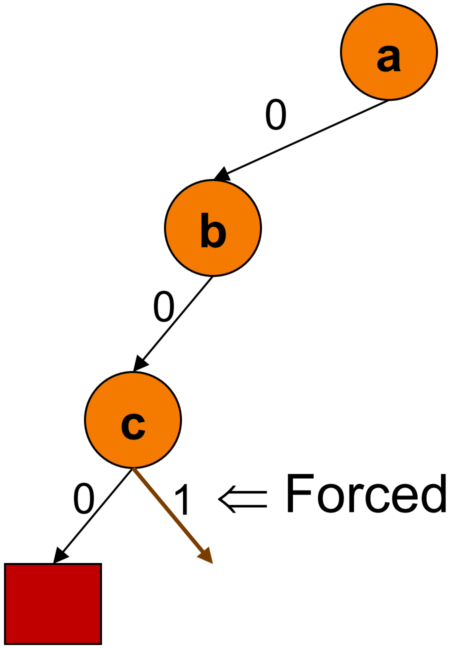
(a’ + b + c’)

(a’ + b’ + c)



Basic DPLL Search

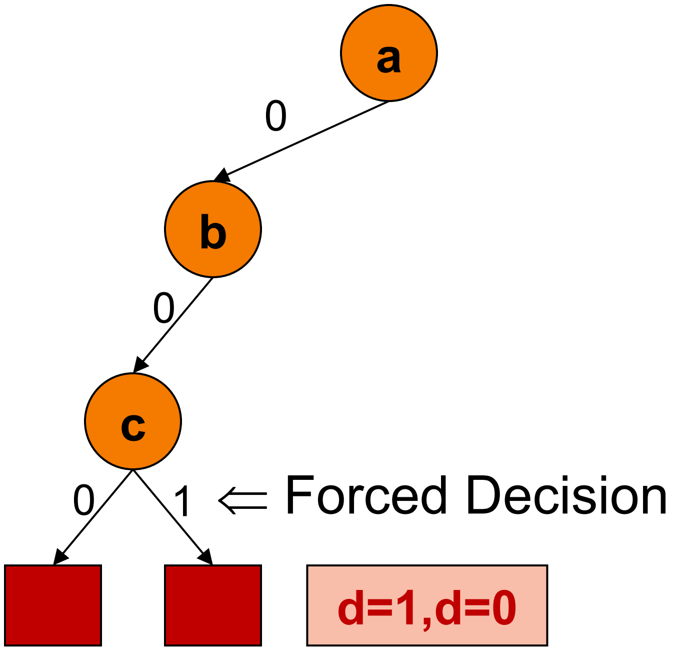
(a’ + b + c)
(a + c + d)
(a + c + d’)
→ (a + c’ + d)
→ (a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



⇐ Forced Decision

Basic DPLL Search

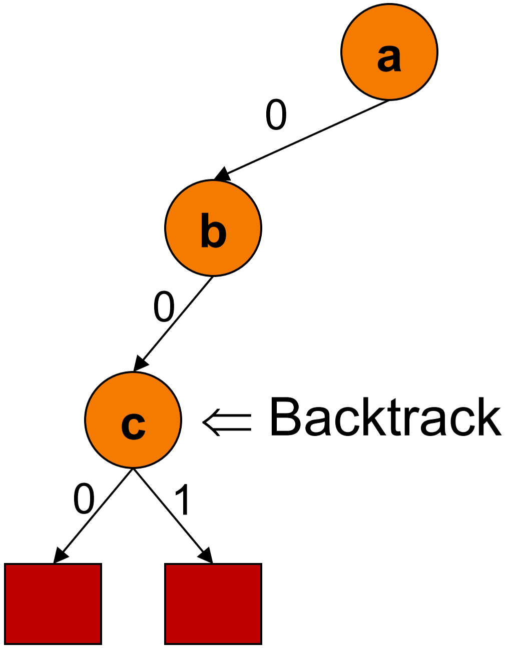
(a’ + b + c)
(a + c + d)
(a + c + d’)
→ (a + c’ + d)
→ (a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



Conflict!

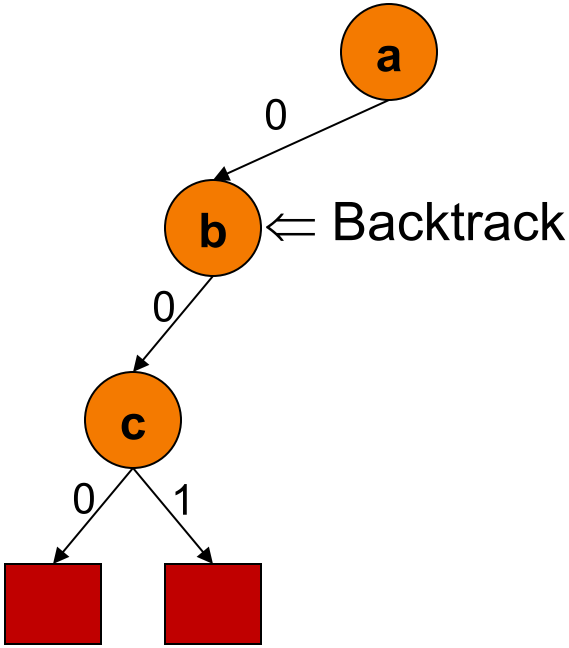
Basic DPLL Search

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



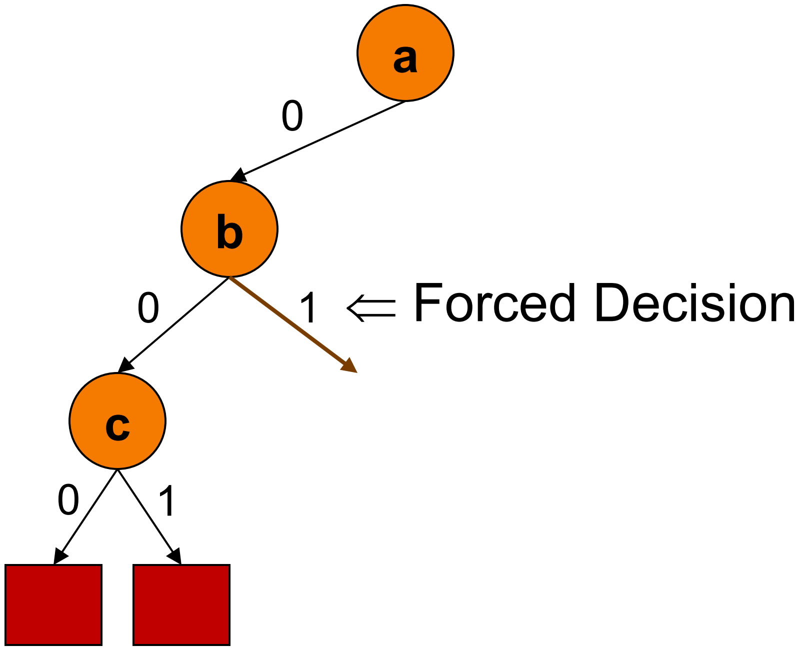
Basic DPLL Search

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



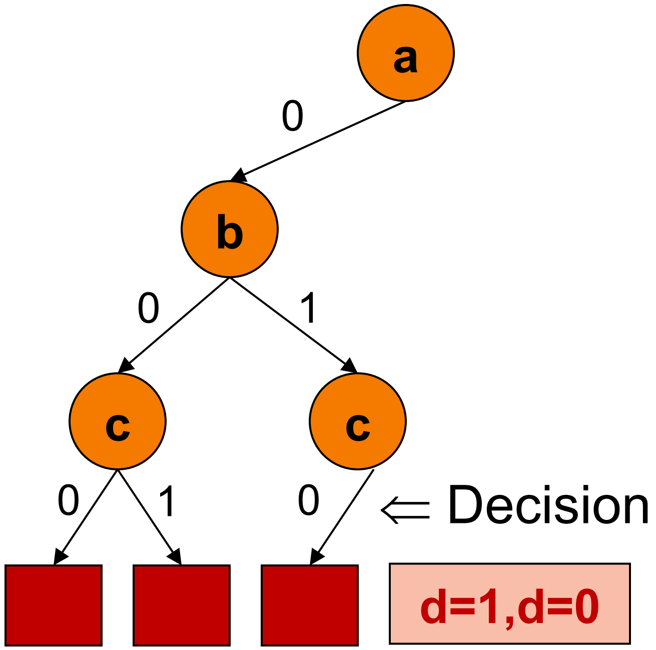
Basic DPLL Search

(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 DPLL Search

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

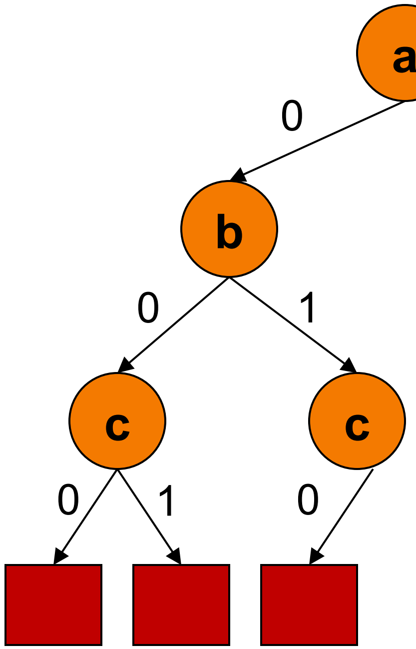


Conflict!

Note: same two clauses are unit (as before) cause the same conflict!

Basic DPLL Search

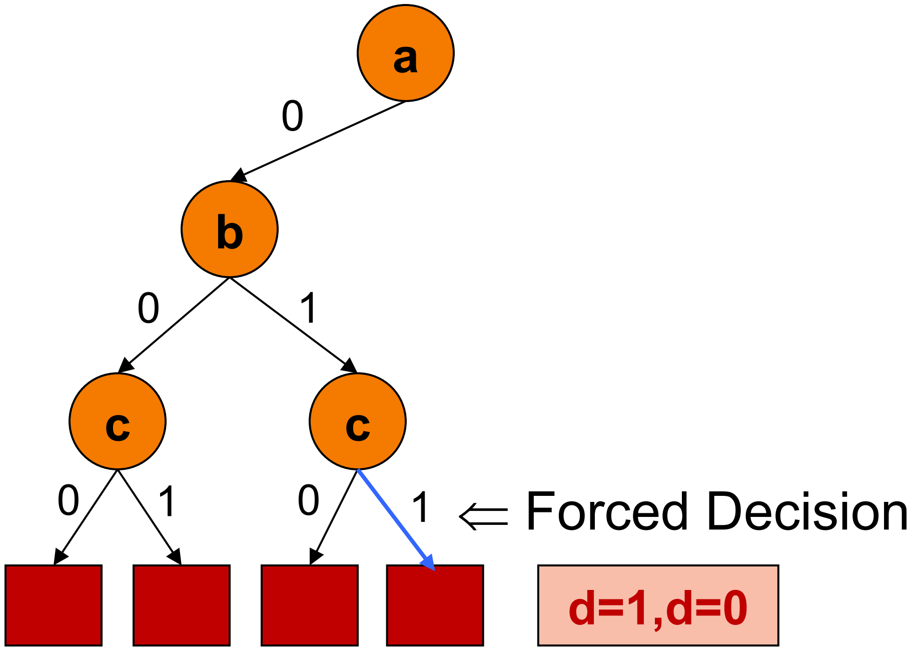
(a’ + b + c)
→ (a + c + d)
→ (a + c + d’)
→ (a + c’ + d)
→ (a + c’ + d’)
→ (b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



⇐ Backtrack

Basic DPLL Search

(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 DPLL Search

(a’ + b + c)

→ (a + c + d)

→ (a + c + d’)

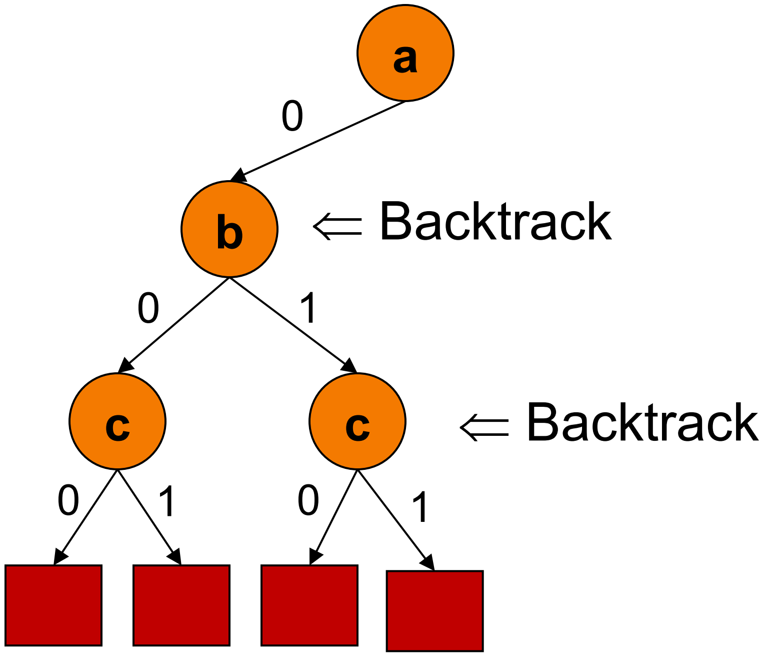
→ (a + c’ + d)

→ (a + c’ + d’)

→ (b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Basic DPLL Search

→ (a’ + b + c)

(a + c + d)

(a + c + d’)

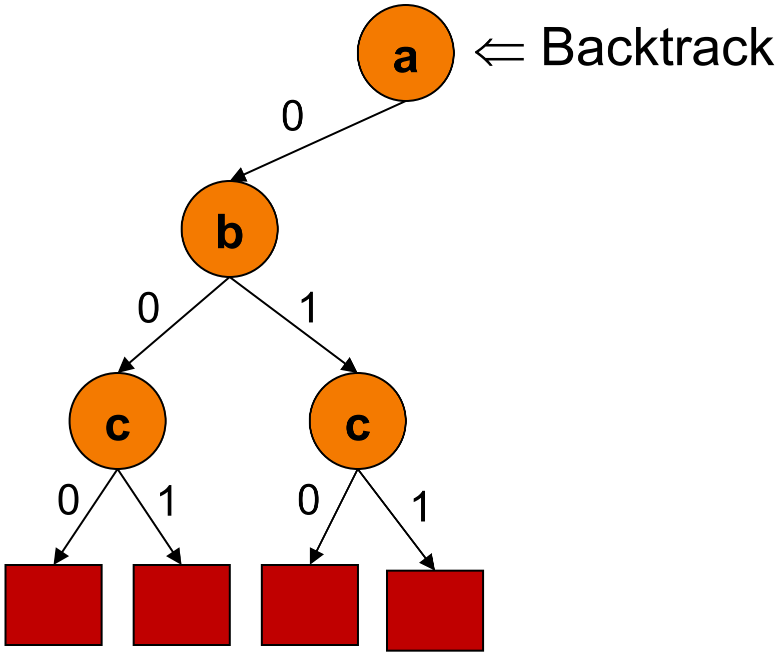
(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)

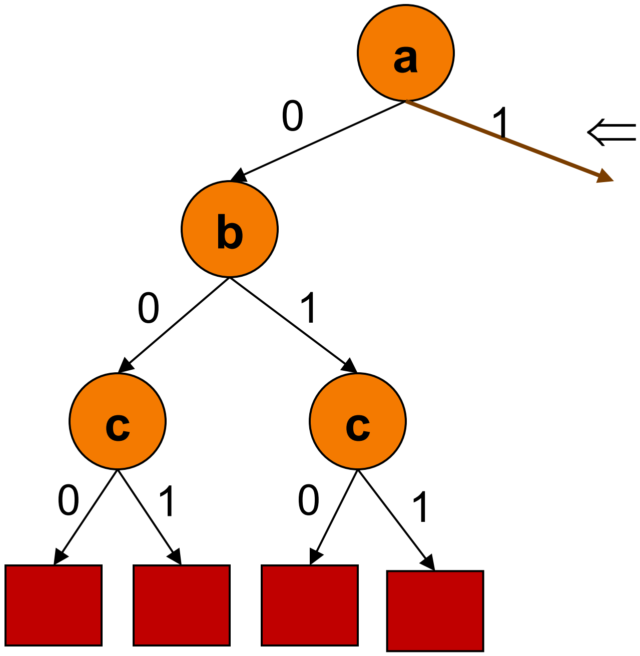
→ (a’ + b + c’)

→ (a’ + b’ + c)



Basic DPLL Search

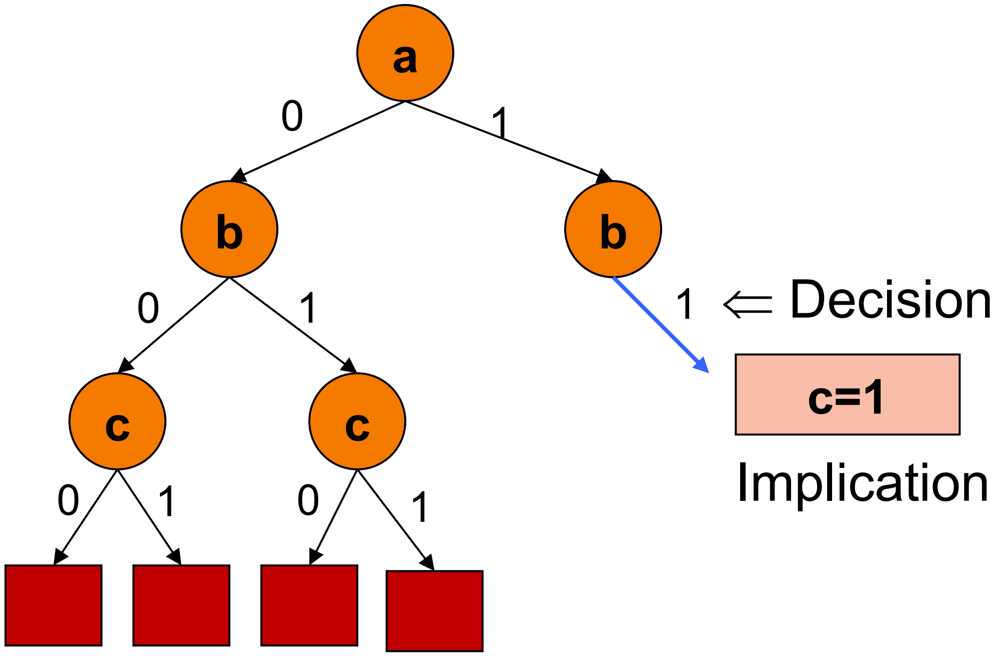
(a’ + b + c)
→ (a + c + d)
→ (a + c + d’)
→ (a + c’ + d)
→ (a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



⇐ Forced Decision

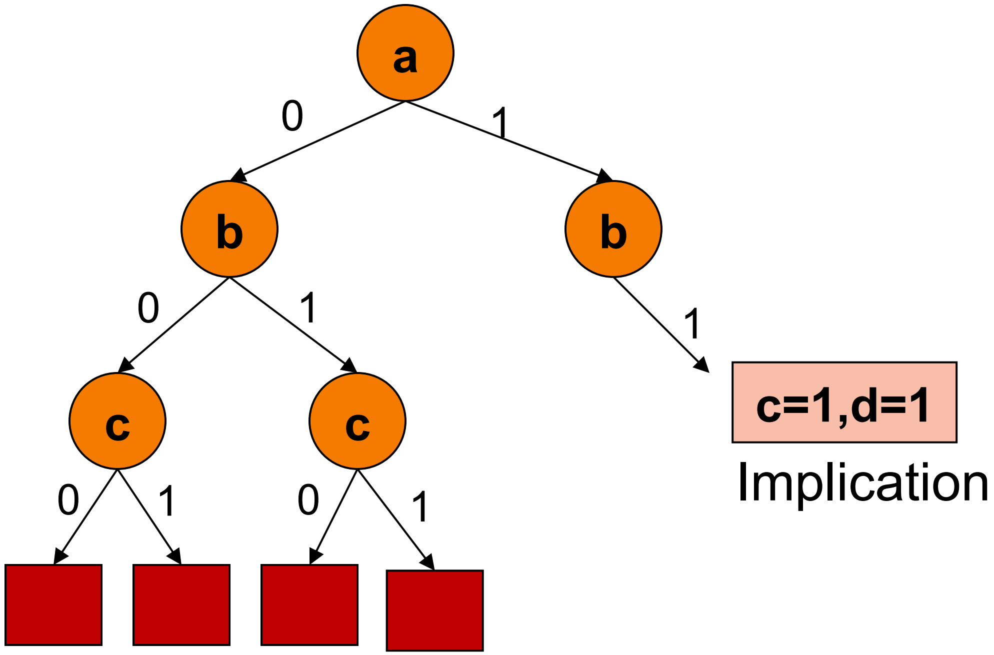
Basic DPLL Search

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



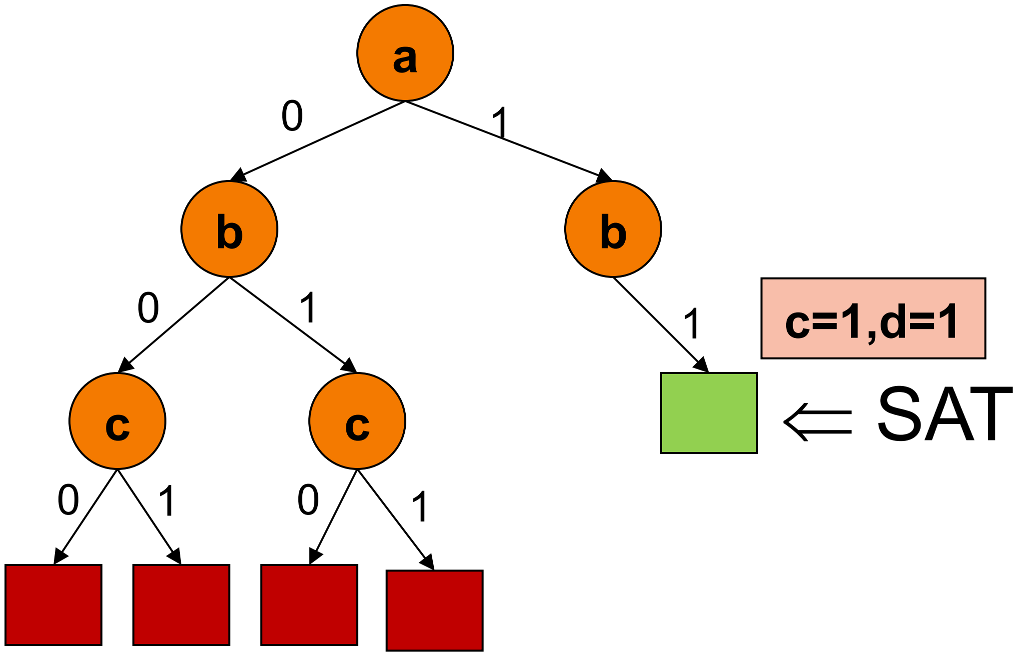
Basic DPLL Search

(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 DPLL Search

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



Backtracking search
with BCP (unit clause rule)

DPLL SAT Solver

unit clause rule

```
DPLL(F)
    G ← **BCP**(F)
    if G = $\top$ then return *true*
    if G = $\bot$ then return *false*
    p ← **choose**(vars(G))
    return DPLL(G{p $\mapsto$ $\top$}) = “SAT” **or** DPLL(G{p $\mapsto$ $\bot$})
```

decision heuristics

backtracking search

Much research, many heuristics over >40 years …

Poor scalability: why?

No learning:
Throws away all the work that concluded that current PA is bad

```
DPLL(F)
    $G \leftarrow$ BCP(F)
    if $G = \top$ then return true
    if $G = \bot$ then return false
    $p \leftarrow$ choose(vars(G))
    return DPLL($G\{p \mapsto \top\}$) = "SAT" or DPLL($G\{p \mapsto \bot\}$)
```

Naïve decision heuristics:
Usually choice is independent of "state" of search

Chronological backtracking:
backtracks one level, even if current PA was doomed at an earlier level

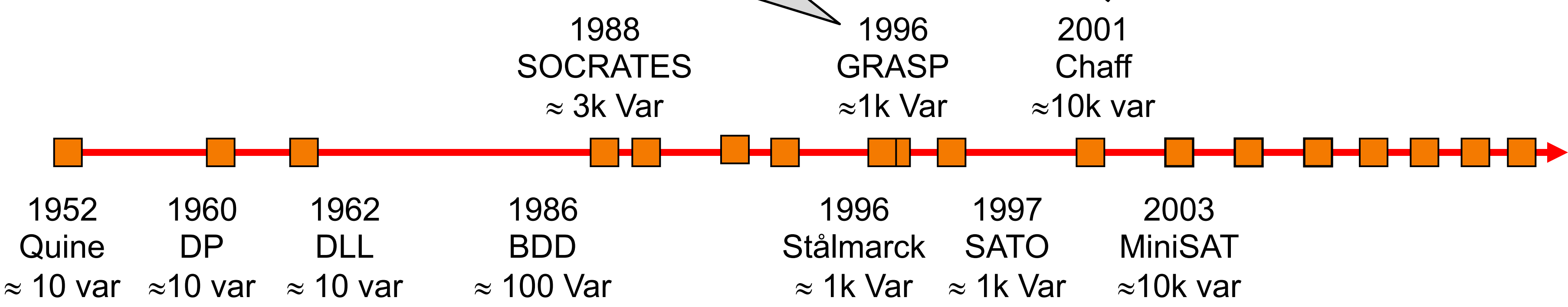
JRW project ideas!

SAT/SMT Timeline

Conflict Driven Clause Learning
Non-chronological backtracking

VSIDS decision heuristic,
2-literal watching in BCP

SA ble



1988
SOCRATES
≈ 3k Var

1996
GRASP
≈1k Var

2001
Chaff
≈10k var

1952
Quine
≈ 10 var

1960
DP
≈10 var

1962
DLL
≈ 10 var

1986
BDD
≈ 100 Var

1996
Stålmarck
≈ 1k Var

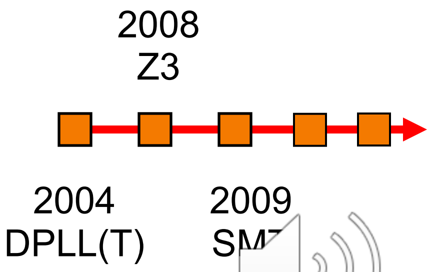
1997
SATO
≈ 1k Var

2003
MiniSAT
≈10k var

(Source: Sharad Malik)

SMT (Satisfiability Modulo Theory):

is a first-order logic formula theory-satisfiable?



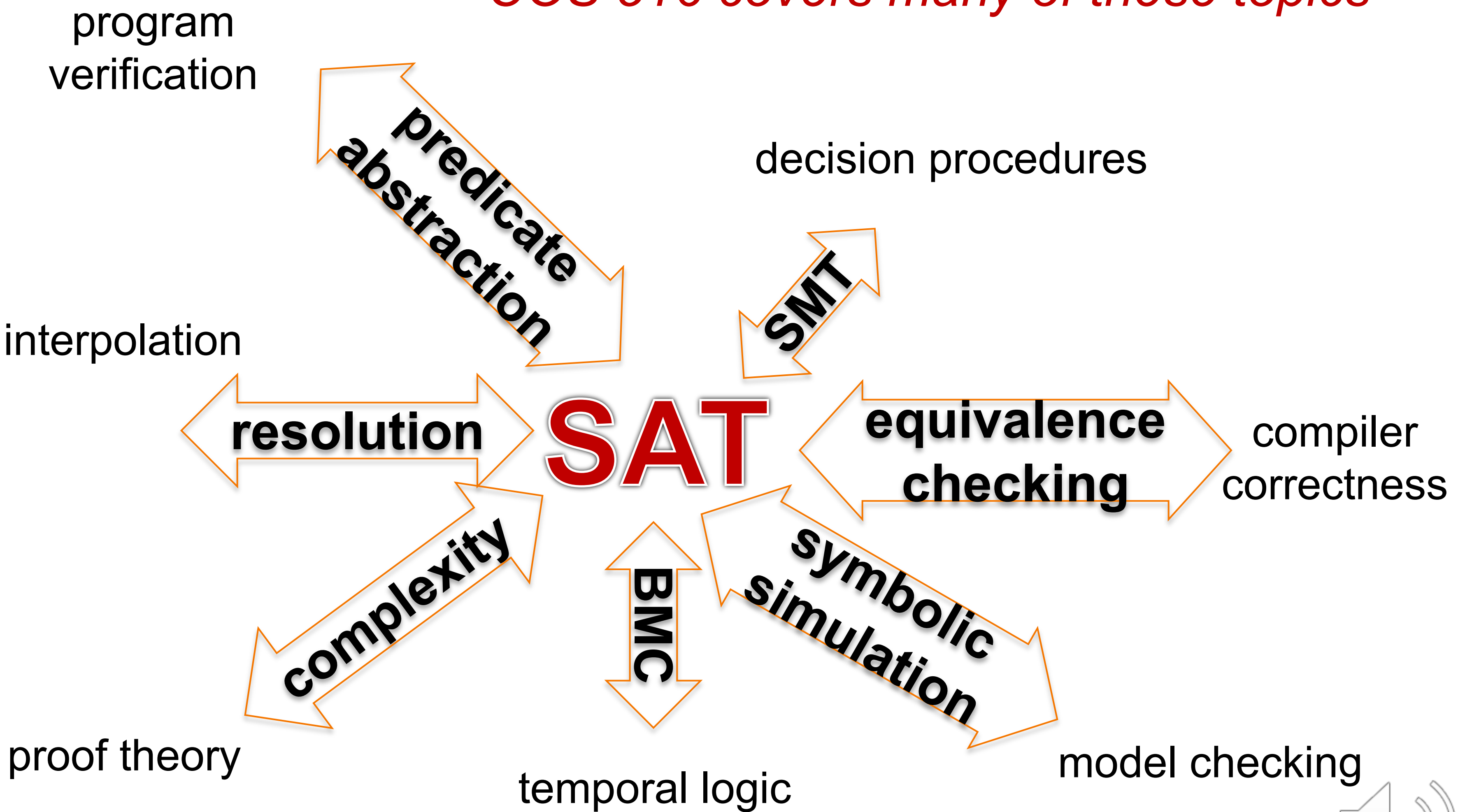
2008
Z3

2004
DPLL(T)

2009
SMT

SAT solvers in verification

COS 516 covers many of these topics



program verification

predicate abstraction

decision procedures

SMT

interpolation

resolution

SAT

equivalence checking

compiler correctness

complexity

BMC

symbolic simulation

proof theory

temporal logic

model checking

Optional Readings

Sharad Malik, Lintao Zhang:
Boolean satisfiability from theoretical hardness to practical success. Communications of the ACM 52(8): 76-82 (2009)

Leonardo Mendonça de Moura, Nikolaj Bjørner:
Satisfiability modulo theories: introduction and applications. Communications of the ACM 54(9): 69-77 (2011)