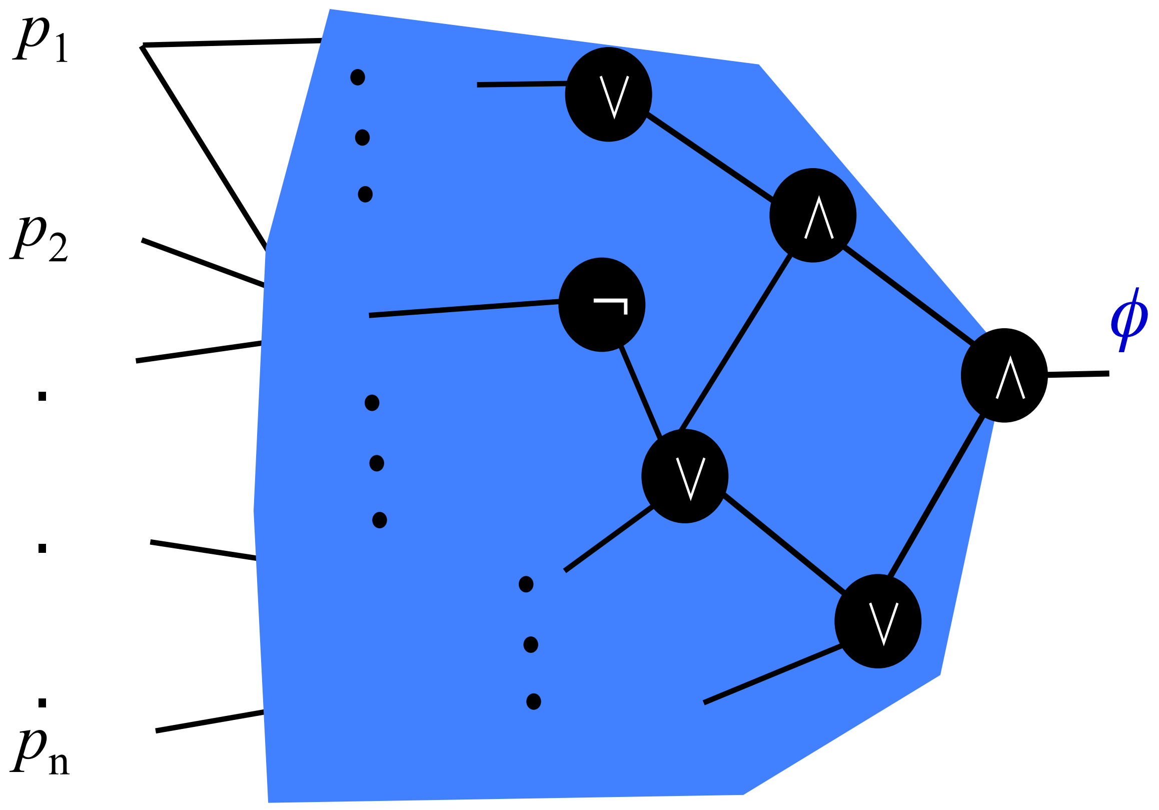
SMT Solvers (an extension of SAT)

Kenneth Roe

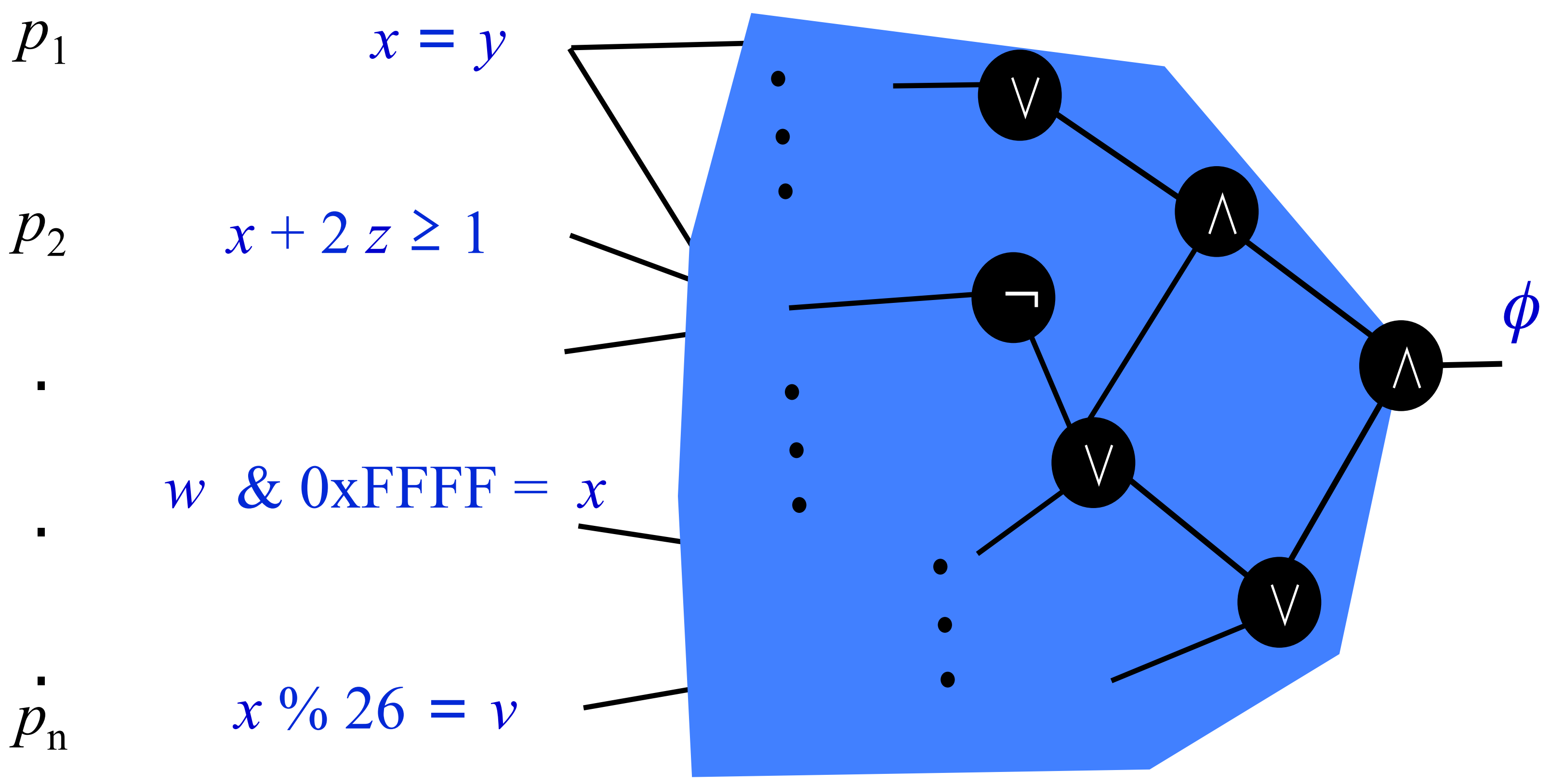
Boolean Satisfiability (SAT)



Is there an assignment to the $p_1, p_2, \ldots, p_n$ variables such that ϕ evaluates to 1?

Slide thanks to C. Barrett & S. A. Seshia, ICCAD 2009 Tutorial

Satisfiability Modulo Theories



Is there an assignment to the x,y,z,w variables s.t. ϕ evaluates to 1?

Satisfiability Modulo Theories

- Given a formula in first-order logic, with associated background theories, is the formula satisfiable?
  - Yes: return a satisfying solution
  - No [generate a proof of unsatisfiability]

Applications of SMT

- Hardware verification at higher levels of abstraction (RTL and above)
- Verification of analog/mixed-signal circuits
- Verification of hybrid systems
- Software model checking
- Software testing
- Security: Finding vulnerabilities, verifying electronic voting machines, …
- Program synthesis
- Scheduling

Slide thanks to C. Barrett & S. A. Seshia, ICCAD 2009 Tutorial (modified)

References

Satisfiability Modulo Theories

Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli.

Chapter 8 in the Handbook of Satisfiability, Armin Biere, Hans van Maaren, and Toby Walsh, editors, IOS Press, 2009.

(available from our webpages)

SMTLIB: A repository for SMT formulas (common format) and tools (www.smtlib.org)

SMTCOMP: An annual competition of SMT solvers

Roadmap for this Tutorial

- Background and Notation
- Survey of Theories
  - Equality of uninterpreted function symbols
  - Bit vector arithmetic
  - Linear arithmetic
  - Difference logic
  - Array theory
- Combining theories
- Review DLL
- Extending DLL to DPLL(t)

Roadmap for this Tutorial

- ➢ Background and Notation
- Survey of Theories
  - Equality of uninterpreted function symbols
  - Bit vector arithmetic
  - Linear arithmetic
  - Difference logic
  - Array theory
- Combining theories
- Review DLL
- Extending DLL to DPLL(t)

First-Order Logic

- A formal notation for mathematics, with expressions involving
  - Propositional symbols
  - Predicates
  - Functions and constant symbols
  - Quantifiers
- In contrast, propositional (Boolean) logic only involves propositional symbols and operators

First-Order Logic: Syntax

- As with propositional logic, expressions in first-order logic are made up of sequences of symbols.
- Symbols are divided into *logical symbols* and *non-logical symbols* or *parameters*.
- Example:

$$(x = y) \land (y = z) \land (f(z) \rightarrow f(x)+1)$$

First-Order Logic: Syntax

- Logical Symbols
  - Propositional connectives: $\wedge$, $\vee$, $\neg$, $\rightarrow$,…
  - Variables: v1, v2, . . .
  - Quantifiers: $\forall$, $\exists$
- Non-logical symbols/Parameters
  - Equality: =
  - Functions: +, -, %, bit-wise &, f(), concat, …
  - Predicates: $\succeq$, is_substring, …
  - Constant symbols: 0, 1.0, null, …

Quantifier-free Subset

- We will largely restrict ourselves to formulas without quantifiers ($\forall$, $\exists$)
- This is called the quantifier-free subset/ fragment of first-order logic with the relevant theory

Logical Theory

- Defines a set of parameters (non-logical symbols) and their meanings
- This definition is called a *signature*.
- Example of a signature:

  Theory of linear arithmetic over integers

  Signature is (0,1,+,-,$\succeq$) interpreted over $\mathbb{Z}$

Roadmap for this Tutorial

- Background and Notation
- Survey of Theories
  - Equality of uninterpreted function symbols
  - Bit vector arithmetic
  - Linear arithmetic
  - Difference logic
  - Array theory
- Review DLL
- Extending DLL to DPLL(t)
- Combining theories

Some Useful Theories

- Equality (with uninterpreted functions)
- Linear arithmetic (over $\mathbb{Q}$ or $\mathbb{Z}$)
- Difference logic (over $\mathbb{Q}$ or $\mathbb{Z}$)
- Finite-precision bit-vectors
  - integer or floating-point
- Arrays / memories
- Misc.: Non-linear arithmetic, strings, inductive datatypes (e.g. lists), sets, …

Slide thanks to C. Barrett & S. A. Seshia, ICCAD 2009 Tutorial

Decision procedure

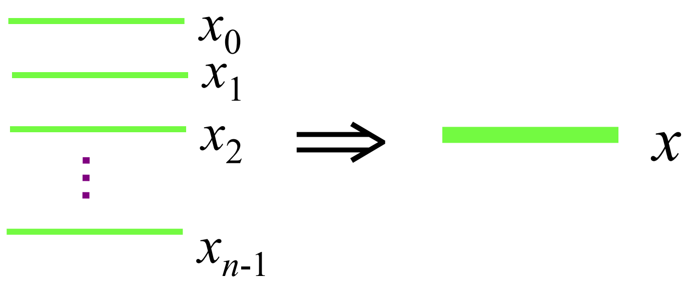
- For each theory there is a decision procedure
- Given a set of predicates in the theory, the procedure will always tell you whether or not they can be satisfied

Theory of Equality and Uninterpreted Functions (EUF)

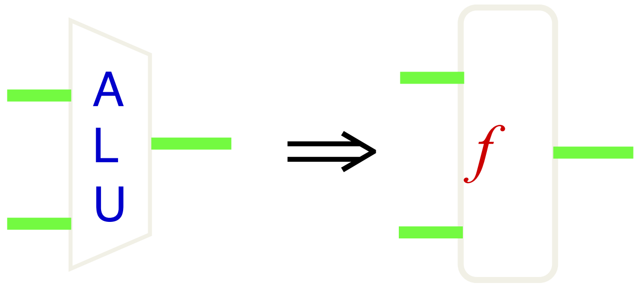
- Also called the "free theory"
  - Because function symbols can take any meaning
  - Only property required is *congruence*: that these symbols map identical arguments to identical values i.e., $x = y \Rightarrow f(x) = f(y)$
- SMTLIB name: QF_UF

Slide thanks to C. Barrett & S. A. Seshia, ICCAD 2009 Tutorial

Data and Function Abstraction with EUF



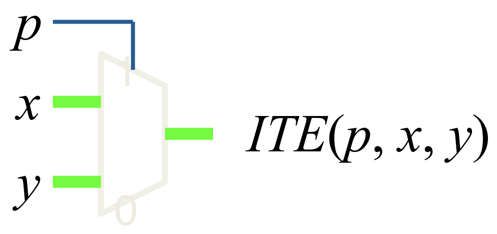
Bit-vectors to Abstract Domain (e.g. $\mathbb{Z}$)



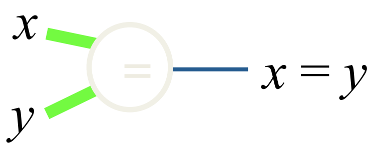
Functional units to Uninterpreted Functions

$$a = x \wedge b = y \Rightarrow f(a,b) = f(x,y)$$

Common Operations

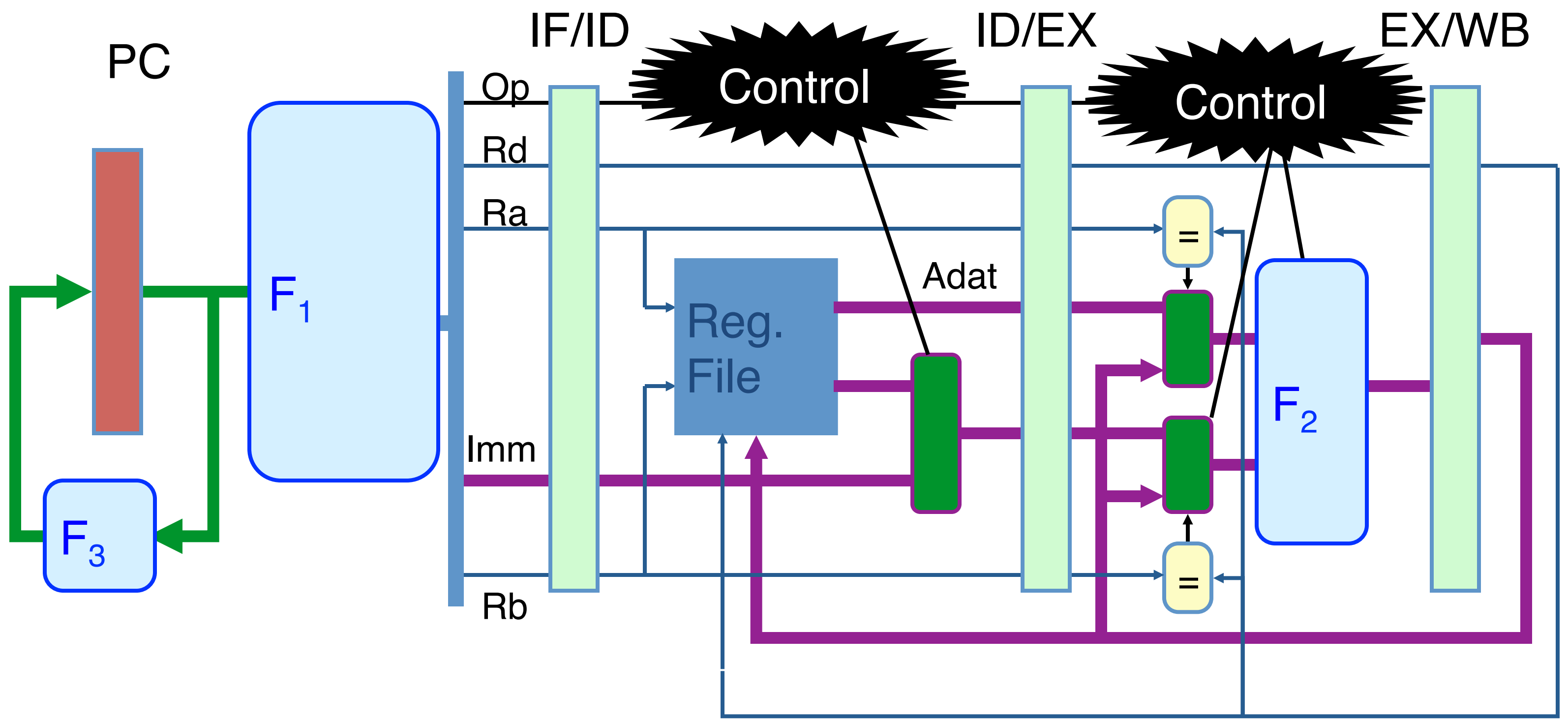


If-then-else



Test for equality

Hardware Abstraction with EUF



- For any Block that Transforms or Evaluates Data:
  - Replace with generic, unspecified function
  - Also view instruction memory as function

Example QF_UF (EUF) Formula

$(x = y) \land (y = z) \land (f(x) \neq f(z))$

Transitivity:

$(x = y) \land (y = z) \rightarrow (x = z)$

Congruence:

$(x = z) \rightarrow (f(x) = f(z))$

Equivalence Checking of Program Fragments

```
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;

   return x*x;
}

int fun2(int y) {
     return y*y;
}
```

SMT formula ϕ

Satisfiable iff programs non-equivalent

$$(z = y \wedge y1 = x \wedge x1 = z \wedge ret1 = x1*x1)$$
$$\wedge$$
$$(ret2 = y*y)$$
$$\wedge$$
$$(ret1 \neq ret2)$$

What if we use SAT to check equivalence?

Equivalence Checking of Program Fragments

```
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;

   return x*x;
}
```

```
int fun2(int y) {
     return y*y;
}
```

SMT formula ϕ
Satisfiable iff programs non-equivalent

$(z = y \pm y1 = x \pm x1 = z \pm ret1 = x1*x1)$
$\pm$
$(ret2 = y*y)$
$\pm$
$(ret1 \neq ret2)$

Using SAT to check equivalence (w/ Minisat)
32 bits for y: Did not finish in over 5 hours
16 bits for y: 37 sec.
8 bits for y: 0.5 sec.

Equivalence Checking of Program Fragments

```
int fun1(int y) {
    int x, z;
    z = y;
    y = x;
    x = z;

   return x*x;
}

int fun2(int y) {
     return y*y;
}
```

SMT formula ϕ'

$$(z = y \wedge y1 = x \wedge x1 = z \wedge ret1 = sq(x1))$$
$$\wedge$$
$$(ret2 = sq(y))$$
$$\wedge$$
$$(ret1 \neq ret2)$$

Using EUF solver: 0.01 sec

Equivalence Checking of Program Fragments

```
int fun1(int y) {
    int x;
    x = x ^ y;
    y = x ^ y;
    x = x ^ y;

  return x*x;
}

int fun2(int y) {
     return y*y;
}
```

Does EUF still work?

No!
Must reason about bit-wise XOR.

Need a solver for bit-vector arithmetic.

Solvable in less than a sec. with a current bit-vector solver.

Finite-Precision Bit-Vector Arithmetic (QF_BV)

- Fixed width data words
  - Can model int, short, long, etc.
- Arithmetic operations
  - E.g., add/subtract/multiply/divide & comparisons
  - Two's complement and unsigned operations
- Bit-wise logical operations
  - E.g., and/or/xor, shift/extract and equality
- Boolean connectives

Linear Arithmetic (QF_LRA, QF_LIA)

- Boolean combination of linear constraints of the form

  $(a_1\, x_1 + a_2\, x_2 + \ldots + a_n\, x_n \gg b)$
- x_i' s could be in $\mathbb{Q}$ or $\mathbb{Z}$, $\gg\ \in\ \{„,>,\succeq,<,=\}$

- Many applications, including:
  - Verification of analog circuits
  - Software verification, e.g., of array bounds

Difference Logic (QF_IDL, QF_RDL)

- Boolean combination of linear constraints of the form

  $x_{\boldsymbol{i}} - x_j \gg c_{ij}$ or $x_i \gg c_i$

  $\gg \ \in \ \{\leq, >, \succeq, <, =\}$, x_i's in $\mathbb{Q}$ or $\mathbb{Z}$

- Applications:
  - Software verification (most linear constraints are of this form)
  - Processor datapath verification
  - Job shop scheduling / real-time systems
  - Timing verification for circuits

Arrays/Memories

- SMT solvers can also be very effective in modeling data structures in software and hardware
  - Arrays in programs
  - Memories in hardware designs: e.g. instruction and data memories, CAMs, etc.

Theory of Arrays (QF_AX) Select and Store

- Two interpreted functions: select and store
  - select(A,i)   Read from A at index i
  - store(A,i,d)   Write d to A at index i
- Two main axioms:
  - select(store(A,i,d), i) = d
  - select(store(A,i,d), j) = select(A,j) for $i \neq j$
- One other axiom:
  - $(\forall$ i. select(A,i) = select(B,i)$) \Rightarrow$ A = B

Equivalence Checking of Program Fragments

```
int fun1(int y) {
   int x[2];
   x[0] = y;
   y = x[1];
   x[1] = x[0];

  return x[1]*x[1];
}


 int fun2(int y) {
      return y*y;
 }
```

SMT formula ϕ''

[x1 = store(x,0,y) ± y1 = select(x1,1)
± x2 = store(x1,1,select(x1,0))
± ret1 = sq(select(x2,1))]
±
(ret2 = sq(y))
±
(ret1 ≠ ret2)

Roadmap for this Tutorial

- ➢ Background and Notation
- Survey of Theories
  - Equality of uninterpreted function symbols
  - Bit vector arithmetic
  - Linear arithmetic
  - Difference logic
  - Array theory
- Combining theories
- Review DLL
- Extending DLL to DPLL(t)

Slide thanks to C. Barrett & S. A. Seshia, ICCAD 2009 Tutorial

Combining Theory Solvers

- Theory solvers become much more useful if they can be used together.

  mux_sel = 0 → mux_out = select(regfile, addr)

  mux_sel = 1 → mux_out = ALU(alu0, alu1)

- For such formulas, we are interested in satisfiability with respect to a *combination* of theories.
- Fortunately, there exist methods for combining theory solvers.
- The standard technique for this is the Nelson-Oppen

method [NO79, TH96].

Slide taken from [Barret09 and Haney]

The Nelson-Oppen Method

- Suppose that *T1* and *T2* are theories and that *Sat* 1 is a theory solver for *T1-satisfiability* and *Sat* 2 for *T2-satisfiability*.
- We wish to determine if ϕ is *T1 ∪T2-satisfiable*.

1. Convert ϕ to its *separate form* ϕ1 ∧ ϕ2.
2. Let S be the set of variables shared between ϕ1 and ϕ2.
3. For each *arrangement* D of S:
    1. Run *Sat* 1 on ϕ1 ∪ D .
    2. Run *Sat* 2 on ϕ2 ∪ D.

Slide taken from [Barret09 and Haney]

Combining Theories

- QF_UFLIA

$\phi = 1 \leq x \land x \leq 2 \land f(x) \neq f(1) \land f(x) \neq f(2)$

- We first convert ϕ to a separate form:
- $\phi_{UF} = f(x) \neq f(y) \land f(x) \neq f(z)$
- $\phi_{LIA} = 1 \leq x \land x \leq 2 \land y = 1 \land z = 2$

Slide taken from [Barret09 and Haney]

Combining Theories

- $\phi_{UF} = f(x) \neq f(y) \land f(x) \neq f(z)$
- $\phi_{LIA} = 1 \leq x \land x \leq 2 \land y = 1 \land z = 2$
- *{x, y, z}* can have 5 possible arrangements based on equivalence classes of *x*, *y*, and *z*
  1. Assume All Variables Equal:
     1. *{x = y, x = z, y = z}* inconsistent with ϕ_{UF}
  2. Assume Two Variables Equal, One Different
     1. *{x = y, x ≠ z, y ≠ z}* inconsistent with ϕ_{UF}
     2. *{x ≠ y, x = z, y ≠ z}* inconsistent with ϕ_{UF}
     3. *{x ≠ y, x ≠ z, y = z}* inconsistent with ϕ_{LIA}
  3. Assume All Variables Different:
     1. *{x ≠ y, x ≠ z, y ≠ z}* inconsistent with ϕ_{LIA}

Φ
IS
UNSAT

Slide adopted from [Barret09 and Haney]

Convex theories

- Definition:

  $\Gamma \models_T \bigvee_{i \in I} x_i = y_i$ iff $\Gamma \models_T x_i = y_i$ for some $i \in I$
- Gives much faster combination
  - $O(2^{n*n} \times (T1(n)+T2(n))$ if one or both theories not convex
  - $O(n^3 \times (T1(n) + T2(n)))$ if both are convex

- Non-convex theories:
  - bit vector theories
  - linear integer arithmetic
  - theory of arrays

Stably infinite theories

- A theory is stably infinite if every satisfiable QFF is satisfiable in an infinite model (Leonardo de Moura)
- T2 =DC($\forall$x,y,z.(x=y)$\vee$(x=z)$\vee$(y=z))

Roadmap for this Tutorial

- ➢ Background and Notation
- Survey of Theories
  - – Equality of uninterpreted function symbols
  - – Bit vector arithmetic
  - – Linear arithmetic
  - – Difference logic
  - – Array theory
- Combining theories
- Review DLL
- Extending DLL to DPLL(t)

Slide thanks to C. Barrett & S. A. Seshia, ICCAD 2009 Tutorial

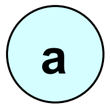
Basic DLL Procedure

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

slide thanks to Sharad Malik (modified)

Basic DLL Procedure

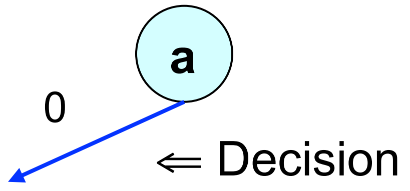
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



Basic DLL Procedure

Green means "crossed out"

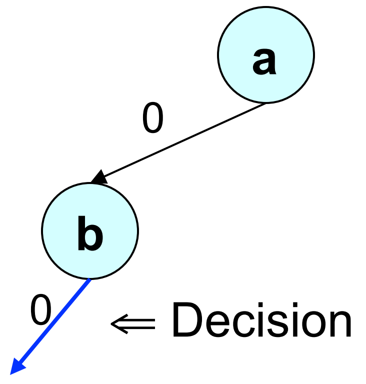
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



slide thanks to Sharad Malik (modified)

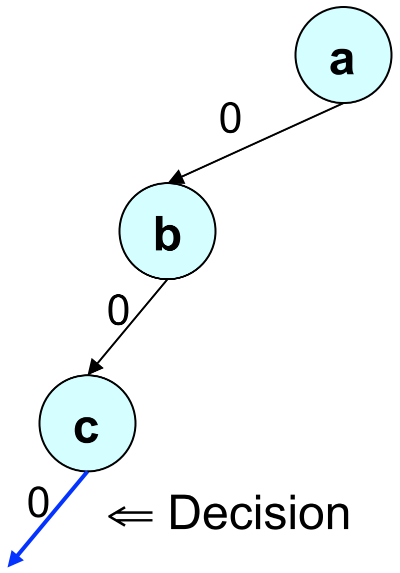
Basic DLL Procedure

(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

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

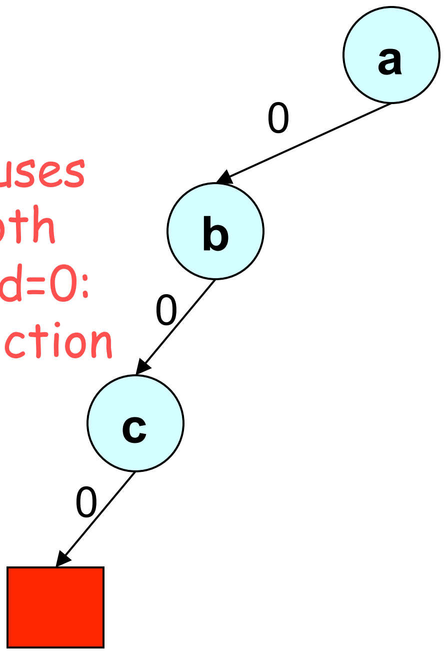


slide thanks to Sharad Malik (modified)

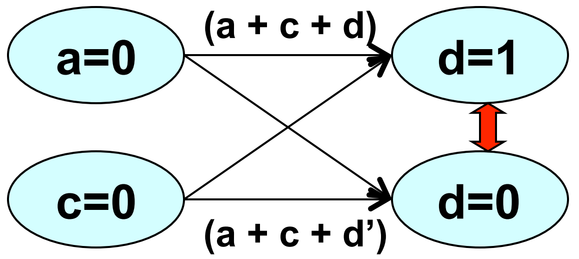
Basic DLL Procedure

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

Unit clauses force both d=1 and d=0: contradiction



Implication Graph (shows that the problem was caused by a=0 ^ c=0; nothing to do with b)

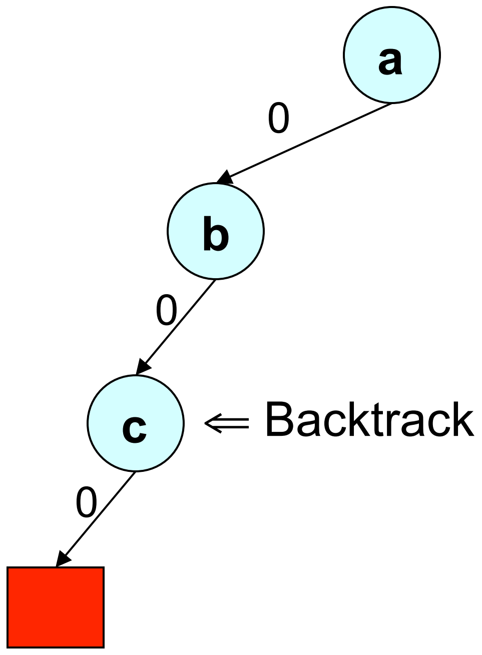


Conflict!

slide thanks to Sharad Malik (modified)

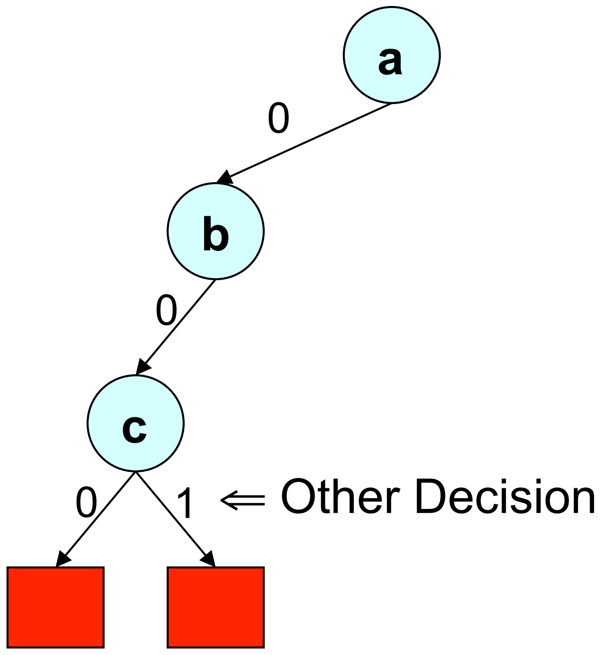
Basic DLL Procedure

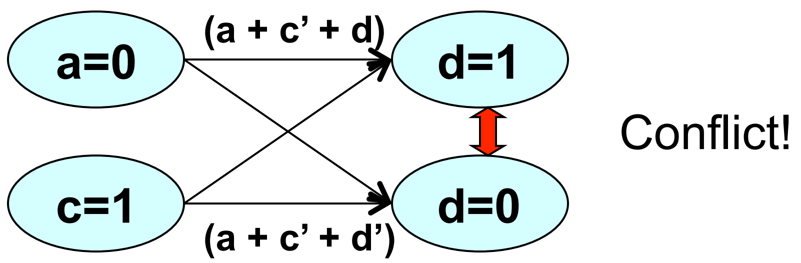
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



Basic DLL Procedure

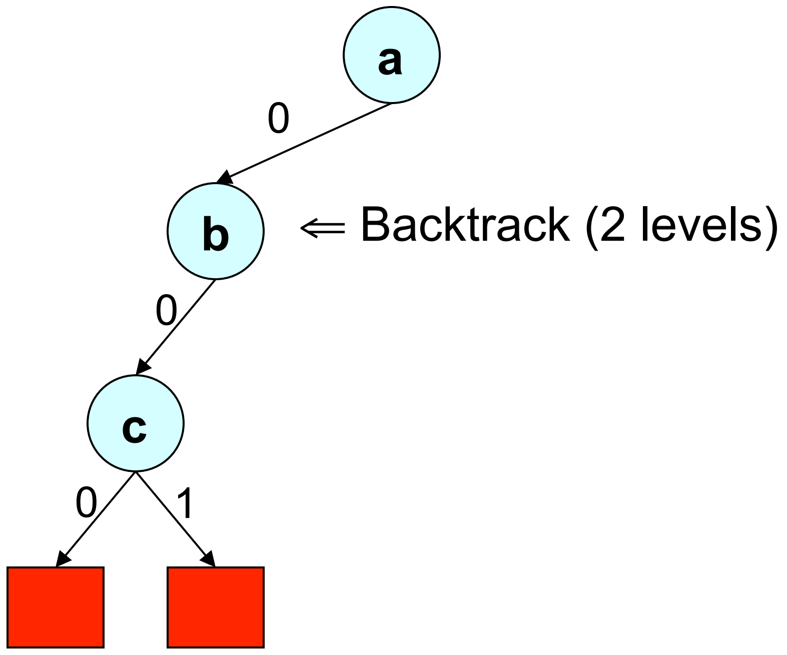
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)





Basic DLL Procedure

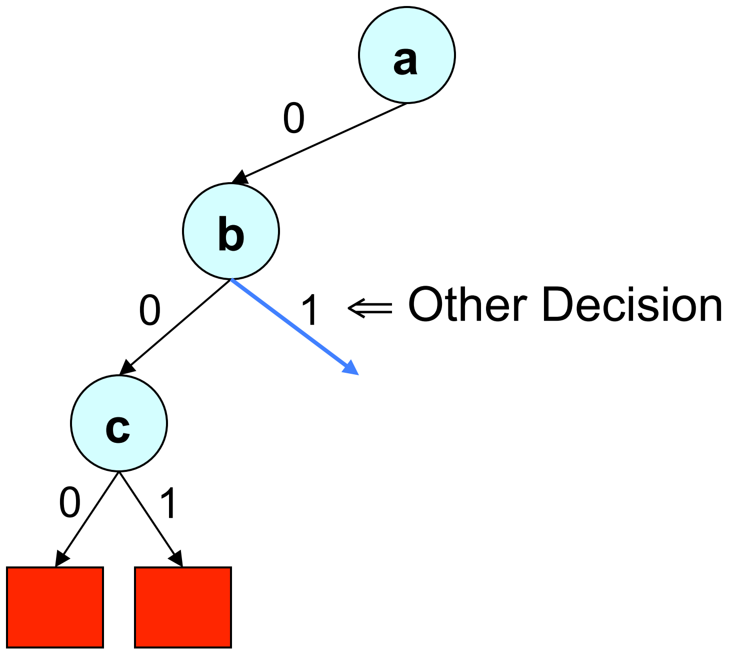
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



slide thanks to Sharad Malik (modified)

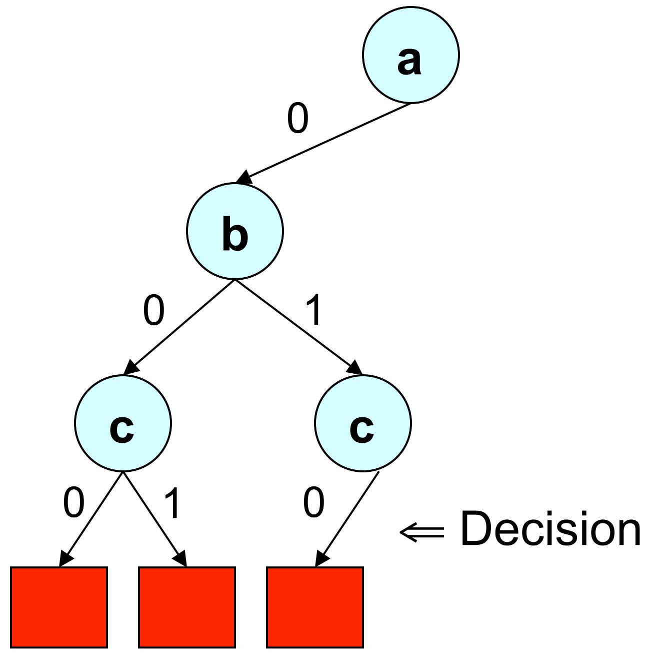
Basic DLL Procedure

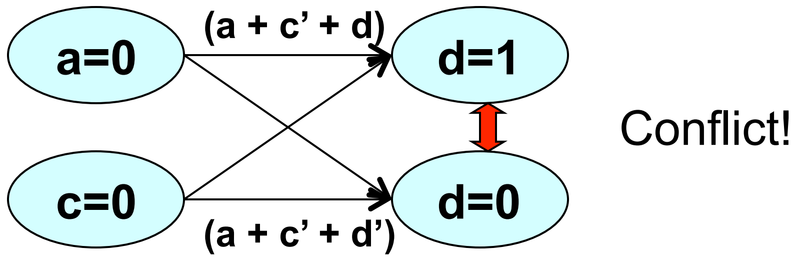
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure

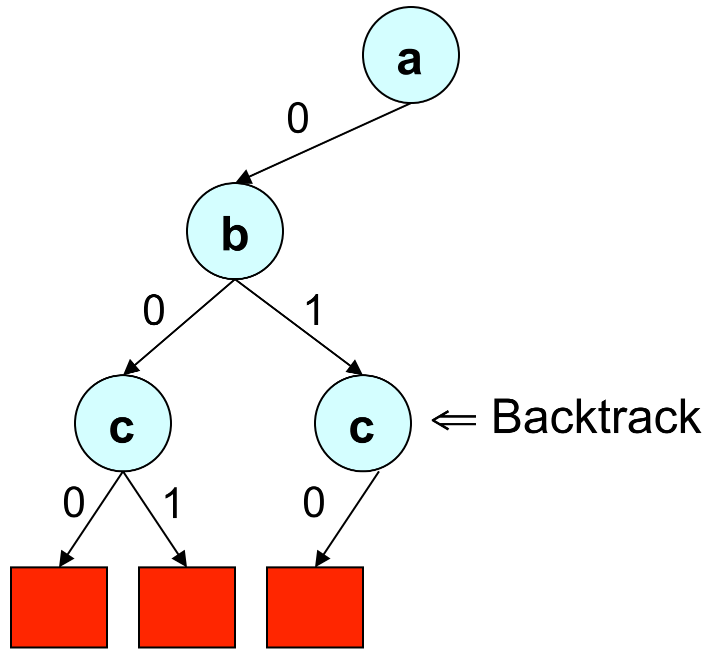
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)





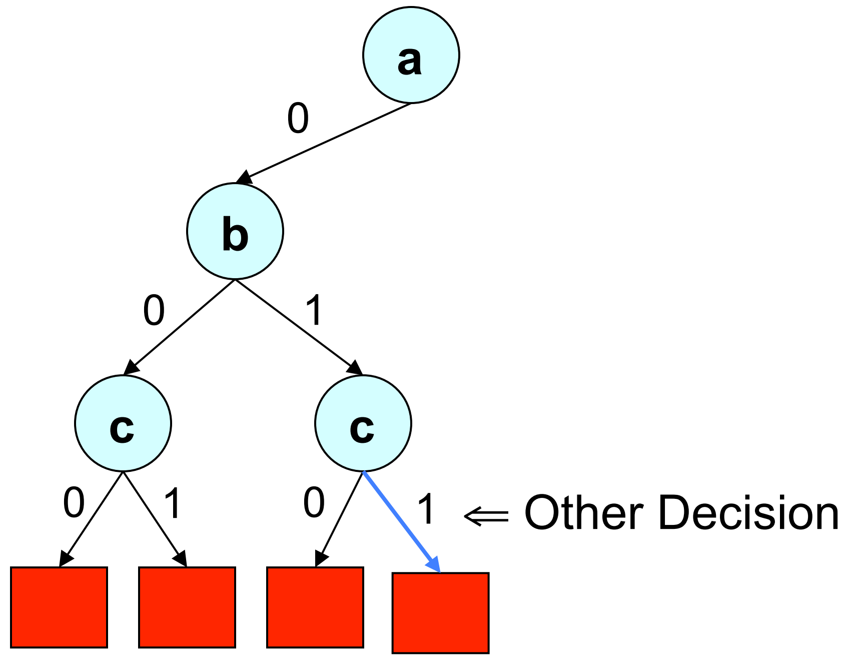
Basic DLL Procedure

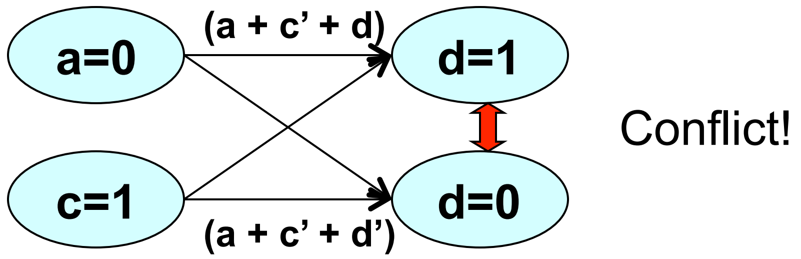
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure

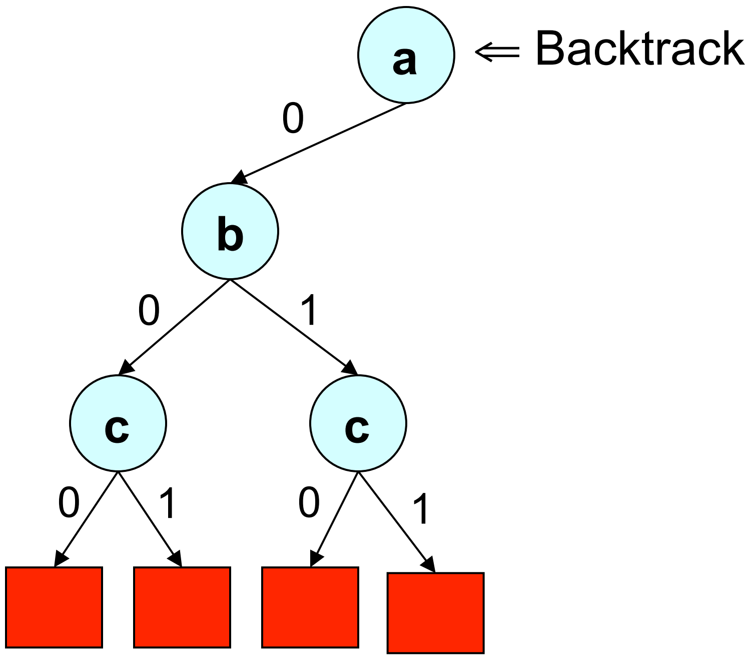
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)





Basic DLL Procedure

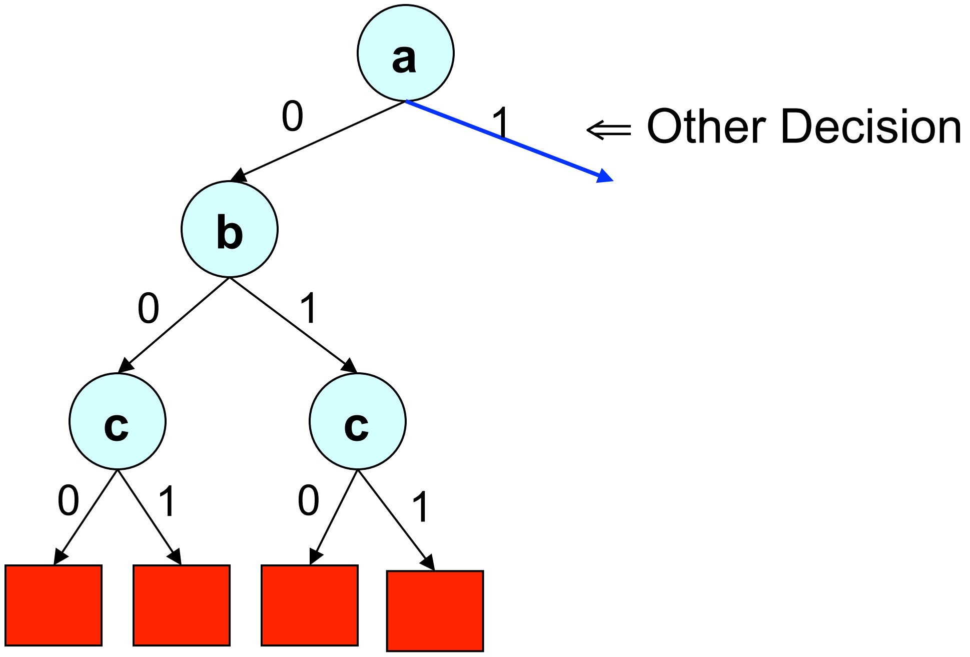
(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)



slide thanks to Sharad Malik (modified)

Basic DLL Procedure

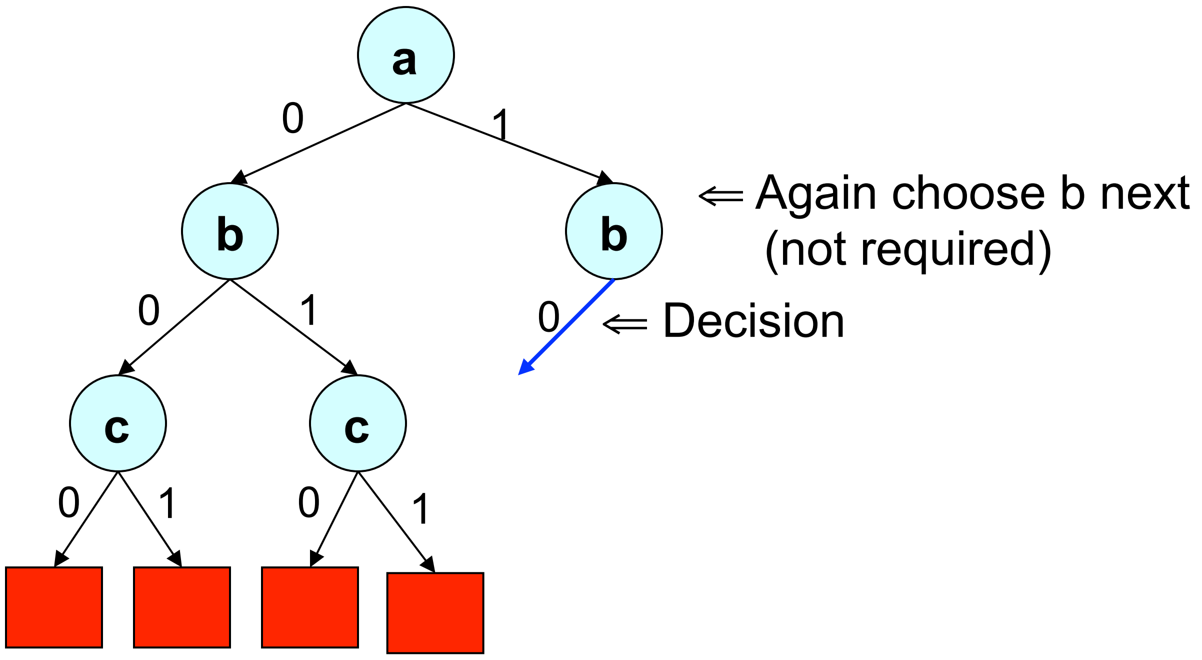
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



slide thanks to Sharad Malik (modified)

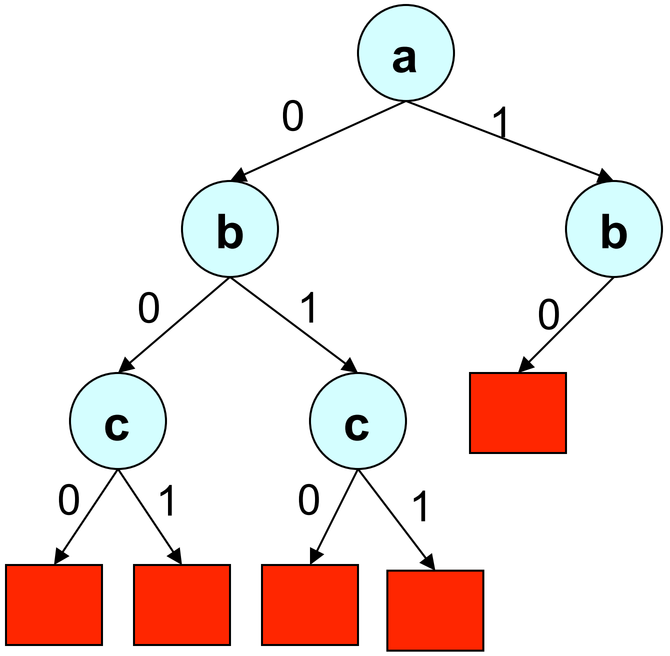
Basic DLL Procedure

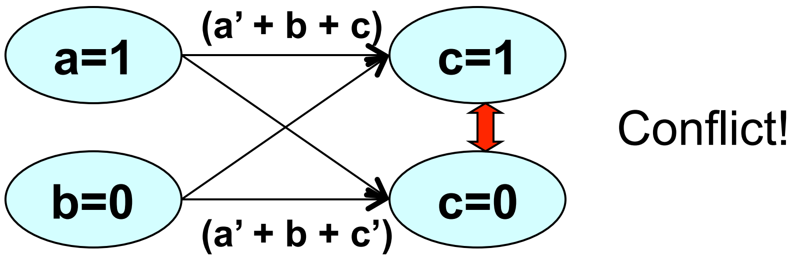
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



Basic DLL Procedure

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

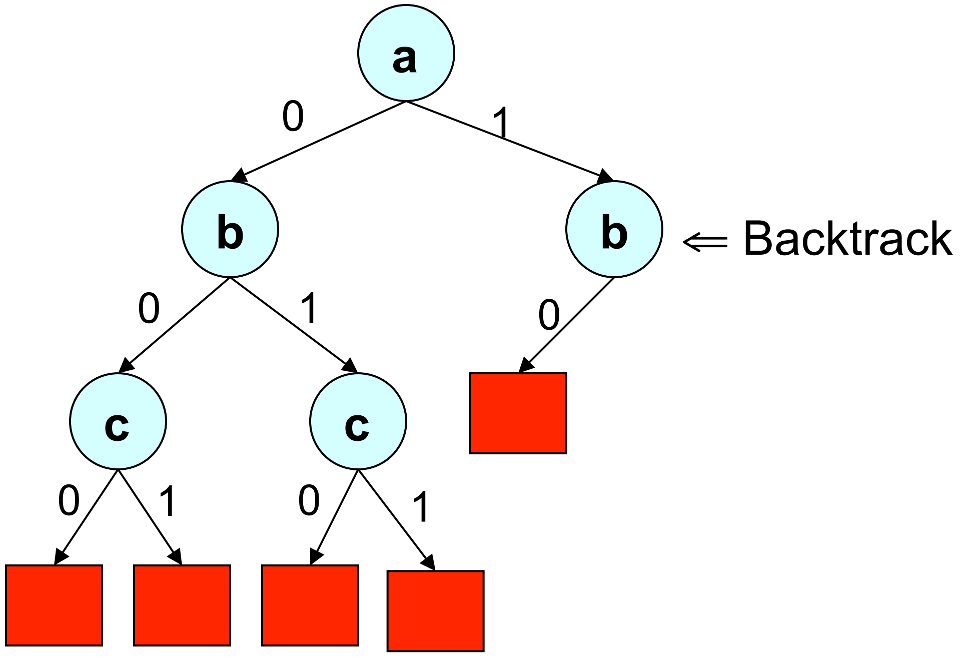




slide thanks to Sharad Malik (modified)

Basic DLL Procedure

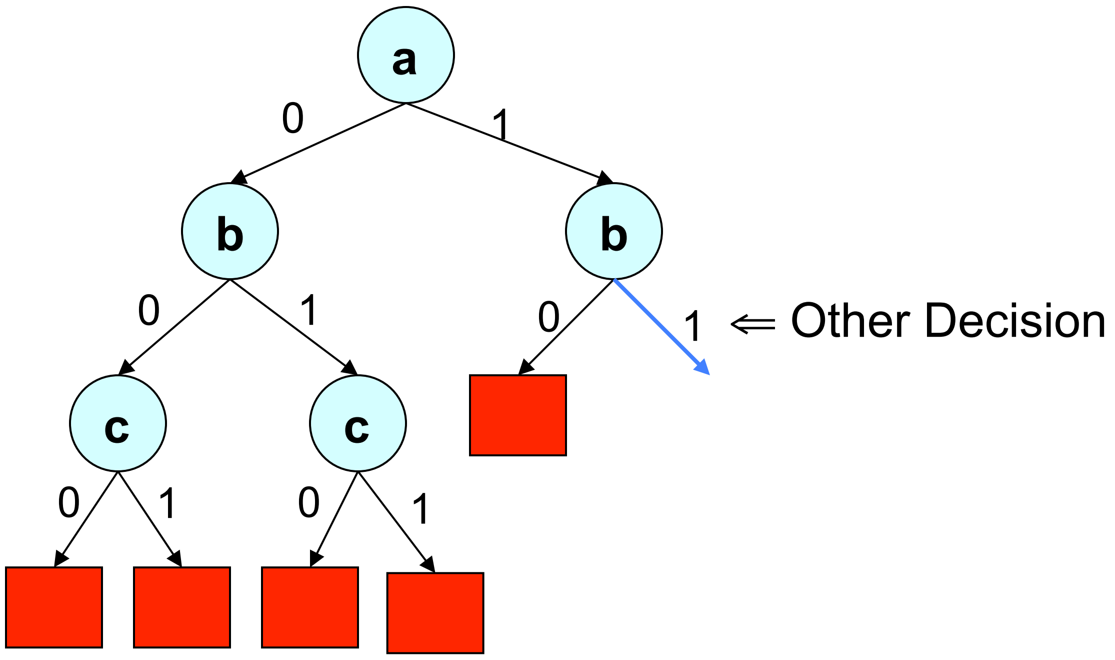
(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)



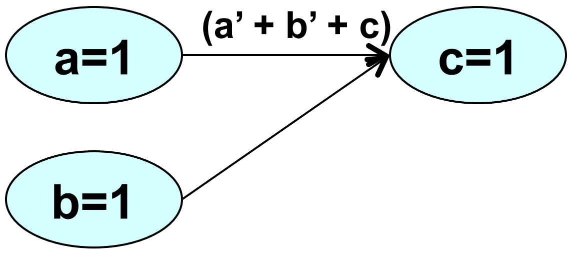
slide thanks to Sharad Malik (modified)

Basic DLL Procedure

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

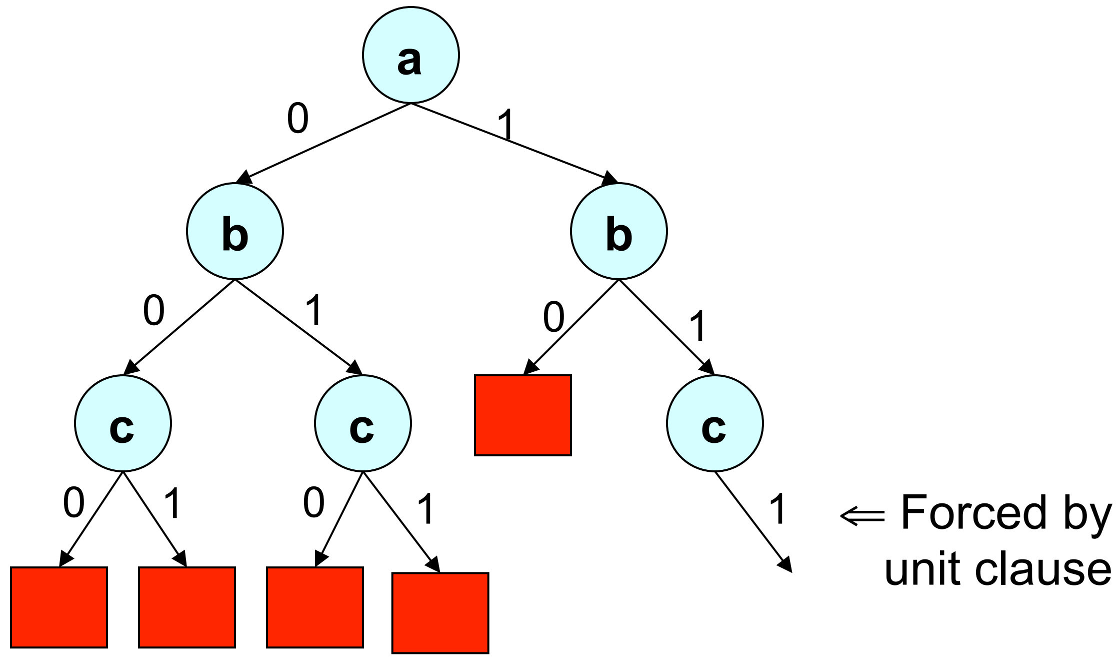


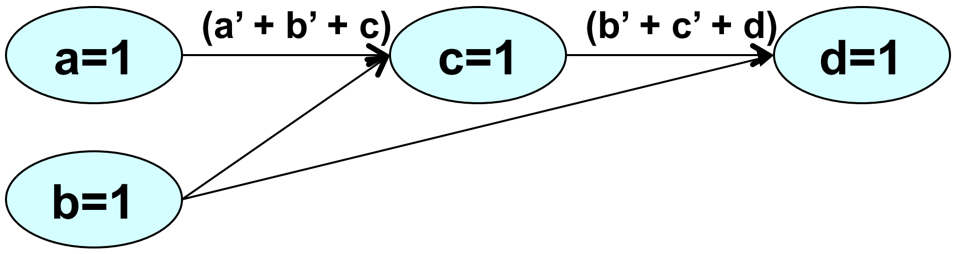
unit clause that propagates without contradiction (finally!)
Often you get these much sooner



Basic DLL Procedure

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

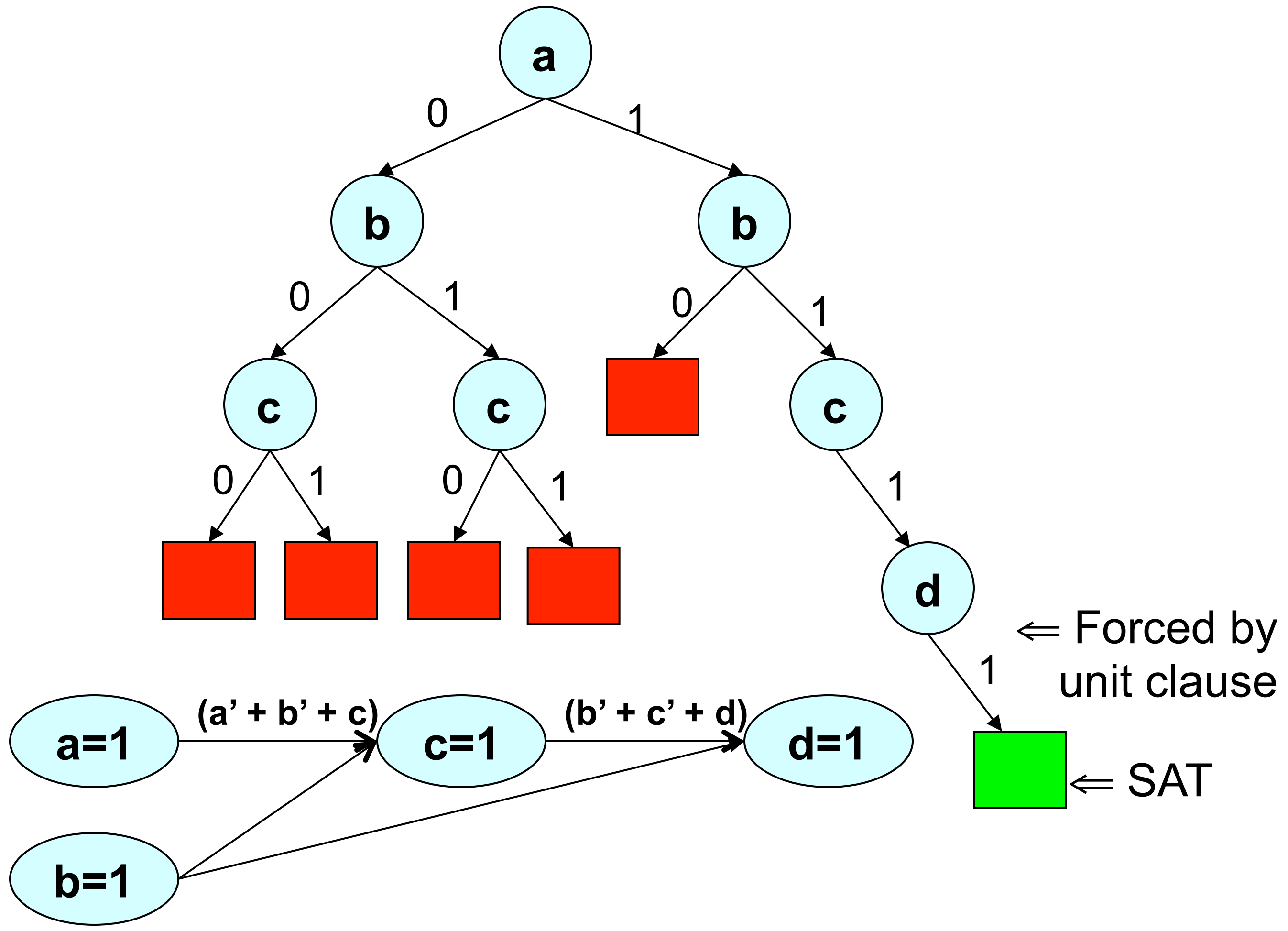




slide thanks to Sharad Malik (modified)

Basic DLL Procedure

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



Tricks used by zChaff and similar DLL solvers

(Overview only; details on later slides)

- **Make unit propagation / backtracking speedy** (80% of the cycles!)
- **Variable ordering heuristics:** Which variable/value to assign next?
- **Conflict analysis:** When a contradiction is found, analyze what subset of the assigned variables was responsible. Why?
  - **Better heuristics:** Like to branch on vars that caused recent conflicts
  - **Backjumping:** When backtracking, avoid trying options that would just lead to the same contradictions again.
  - **Clause learning:** Add new clauses to block bad sub-assignments.
  - **Random restarts** (maybe): Occasionally restart from scratch, but keep using the learned clauses. (Example: crosswords ...)
    - Even without clause learning, random restarts can help by abandoning an unlucky, slow variable ordering. Just break ties differently next time.
- **Preprocess** the input formula (maybe)
- **Tuned implementation:** Carefully tune data structures
  - improve memory locality and avoid cache misses

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

	1dlx_c_mc_ex_bp_f
Num Variables	776
Num Clauses	3725
Num Literals	10045

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

slide thanks to Moskewicz, Madigan, Zhang, Zhao, & Malik

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1		1			0	0						

Stack of assignments used for backtracking

🟨 C=1	🟨 F=0	A=1	G=0							

⬜	= forced by propagation
🟨	= first guess
🟥	= second guess

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1		1			0	0			0			

Stack of assignments used for backtracking

🟨 C=1	🟨 F=0	A=1	G=0	🟨 J=0						

Guess a new assignment J=0

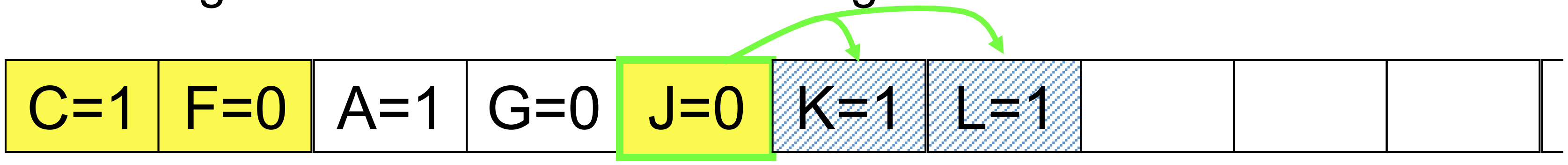
⬜	= forced by propagation
🟨	= first guess
🟥	= second guess

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1		1			0	0			0			

Stack of assignments used for backtracking



Unit propagation implies assignments K=1, L=1

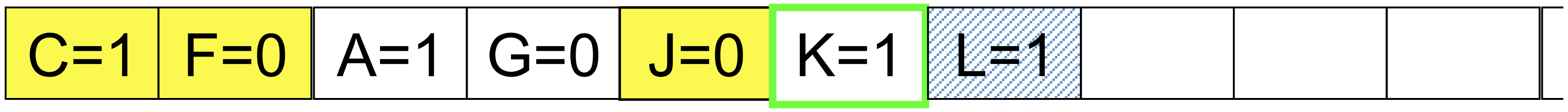
☐	= forced by propagation
🟨	= first guess
🟥	= second guess
🟩	= currently being propagated
▨	= assignment still pending

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1		1			0	0			0	1		

Stack of assignments used for backtracking



Now make those assignments, one at a time

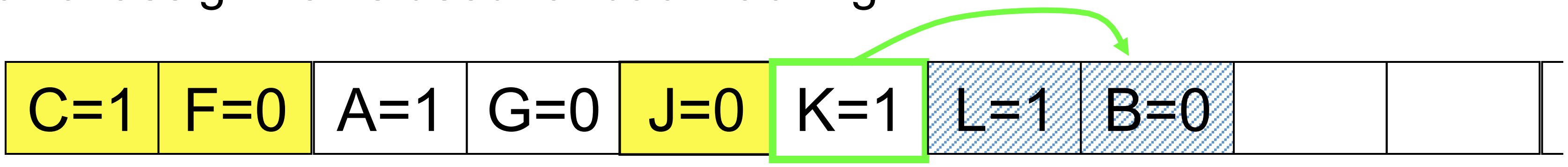
- ☐ = forced by propagation
- 🟨 = first guess
- 🟥 = second guess
- 🟩 = currently being propagated
- ▨ = assignment still pending

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1		1			0	0			0	1		

Stack of assignments used for backtracking



Chain reaction: K=1 propagates to imply B=0

⬜	= forced by propagation
🟨	= first guess
🟥	= second guess
🟩	= currently being propagated
▨	= assignment still pending

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1		1			0	0			0	1		

Stack of assignments used for backtracking

C=1	F=0	A=1	G=0	J=0	K=1	L=1	B=0		

Also implies A=1, but we already knew that

☐	= forced by propagation		
🟨	= first guess	🟩	= currently being propagated
🟥	= second guess	▨	= assignment still pending

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1		1			0	0			0	1	1	

Stack of assignments used for backtracking

C=1	F=0	A=1	G=0	J=0	K=1	L=1	B=0		

⬜ = forced by propagation

🟨 = first guess

🟥 = second guess

🟩 = currently being propagated

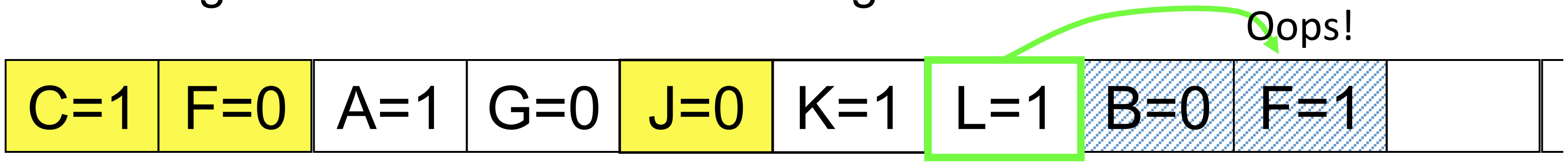
▨ = assignment still pending

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1		1			0	0			0	1	1	

Stack of assignments used for backtracking



L=1 propagates to imply F=1, but we already had F=0

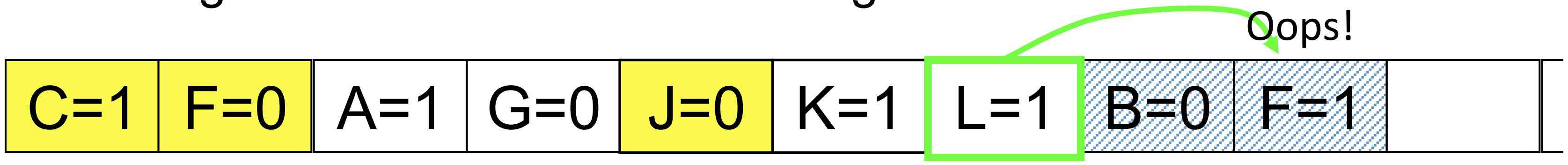
☐	= forced by propagation
🟨	= first guess
🟥	= second guess
🟩	= currently being propagated
▨	= assignment still pending

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1		1			0	0			0	1	1	

Stack of assignments used for backtracking



Backtrack to last yellow, undoing all assignments

⬜	= forced by propagation
🟨	= first guess
🟥	= second guess
🟩	= currently being propagated
▨	= assignment still pending

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1		1			0	0			0			

Stack of assignments used for backtracking

🟨 C=1	🟨 F=0	A=1	G=0	🟨 J=0					

⬜	= forced by propagation		
🟨	= first guess	🟩	= currently being propagated
🟥	= second guess	▨	= assignment still pending

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1		1			0	0			1			

Stack of assignments used for backtracking

🟨 C=1	🟨 F=0	A=1	G=0	🟥 J=1						

J=0 didn't work out, so try J=1

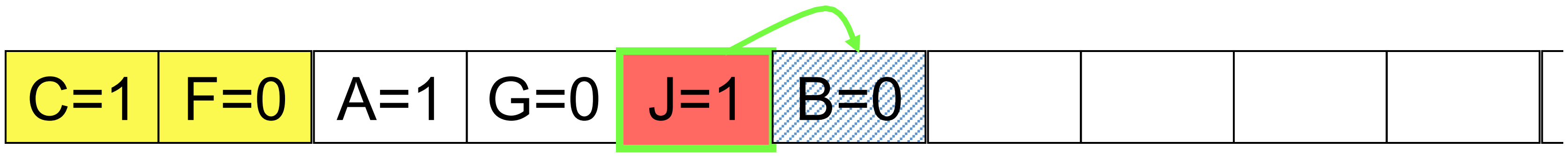
⬜	= forced by propagation
🟨	= first guess
🟥	= second guess
🟩	= currently being propagated
▨	= assignment still pending

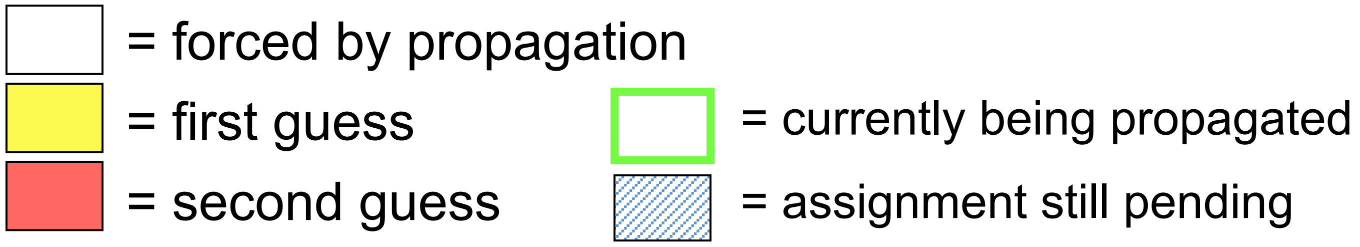
DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1		1			0	0			1			

Stack of assignments used for backtracking





DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1	0	1			0	0			1			

Stack of assignments used for backtracking

C=1	F=0	A=1	G=0	J=1	B=0					

Nothing left to propagate. Now what?

☐	= forced by propagation		
🟨	= first guess	🟩	= currently being propagated
🟥	= second guess	▨	= assignment still pending

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1	0	1			0	0			1		1	

Stack of assignments used for backtracking

🟨 C=1	🟨 F=0	A=1	G=0	🟥 J=1	B=0	🟨 L=1	…			

Again, guess an unassigned variable and proceed …

⬜	= forced by propagation		
🟨	= first guess	🟩	= currently being propagated
🟥	= second guess	▨	= assignment still pending

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1	0	1			0	0			1		0	

Stack of assignments used for backtracking

C=1	F=0	A=1	G=0	J=1	B=0	L=0	…		

If L=1 doesn't work out, we know L=0 in this context

□ = forced by propagation

🟨 = first guess

🟥 = second guess

🟩 = currently being propagated

▨ = assignment still pending

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1	0	1			0	0			1		0	

Stack of assignments used for backtracking

🟨 C=1	🟨 F=0	A=1	G=0	🟥 J=1	B=0	🟥 L=0	…		

If L=0 doesn't work out either, backtrack to … ?

⬜	= forced by propagation
🟨	= first guess
🟥	= second guess
🟩	= currently being propagated
▨	= assignment still pending

DLL: Obvious data structures

Current variable assignments

A	B	C	D	E	F	G	H	I	J	K	L	M
1					1							

Stack of assignments used for backtracking

C=1	F=1									

Question: When should we return SAT or UNSAT?

⬜	= forced by propagation		
🟨	= first guess	🟩	= currently being propagated
🟥	= second guess	▨	= assignment still pending

Roadmap for this Tutorial

- ➢ Background and Notation
- Survey of Theories
  - Equality of uninterpreted function symbols
  - Bit vector arithmetic
  - Linear arithmetic
  - Difference logic
  - Array theory
- Combining theories
- Review DLL
- Extending DLL to DPLL(t)

Basic DPLL(t) Procedure

$p = 3 < x$
$q = x < 0$
$r = x < y$
$s = y < 0$

p
$q \lor r$
$s \lor \sim r$

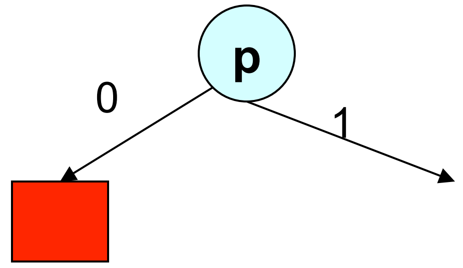
Example, courtesy Leonardo de Moura

Basic DPLL(t) Procedure

Green means "crossed out"

p = 3 < x
q = x < 0
r = x < y
s = y < 0

p
q v r
s v ~r



⇐ Forced by unit clause

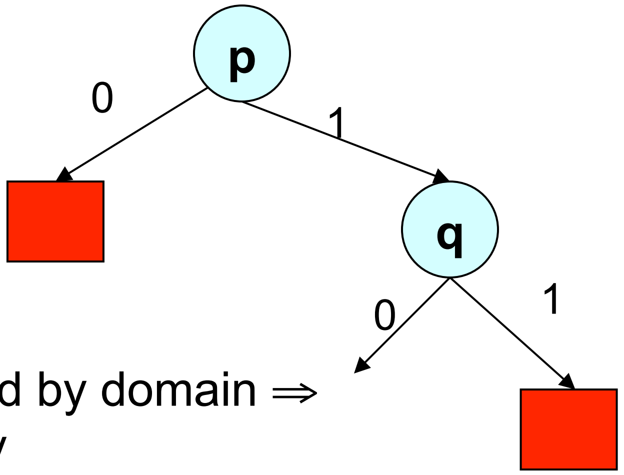
Example, courtesy Leonardo de Moura

Basic DPLL(t) Procedure

Green means "crossed out"

$p = 3 < x$
$q = x < 0$
$r = x < y$
$s = y < 0$

p
q v r
s v ~r



Forced by domain theory ⇒

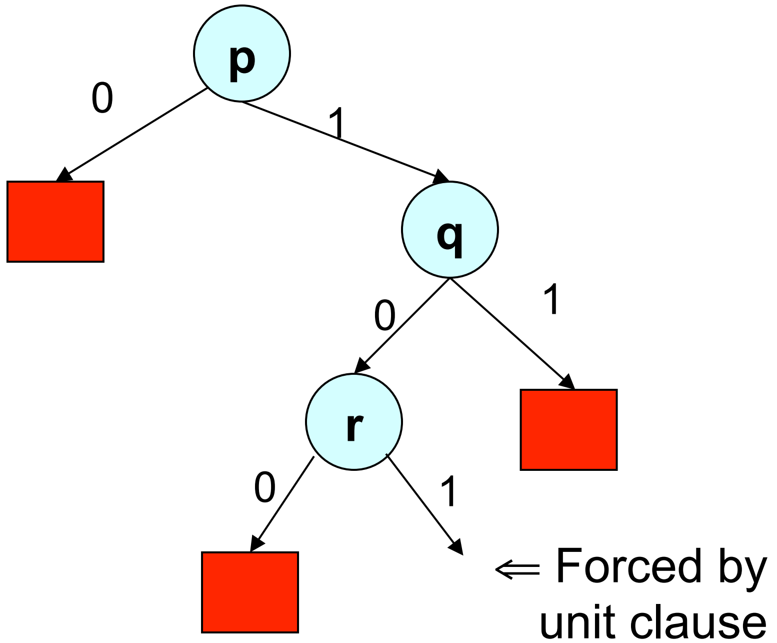
Example, courtesy Leonardo de Moura

Basic DPLL(t) Procedure

Green means "crossed out"

p = 3 < x
q = x < 0
r = x < y
s = y < 0

p
q v r
s v ~r



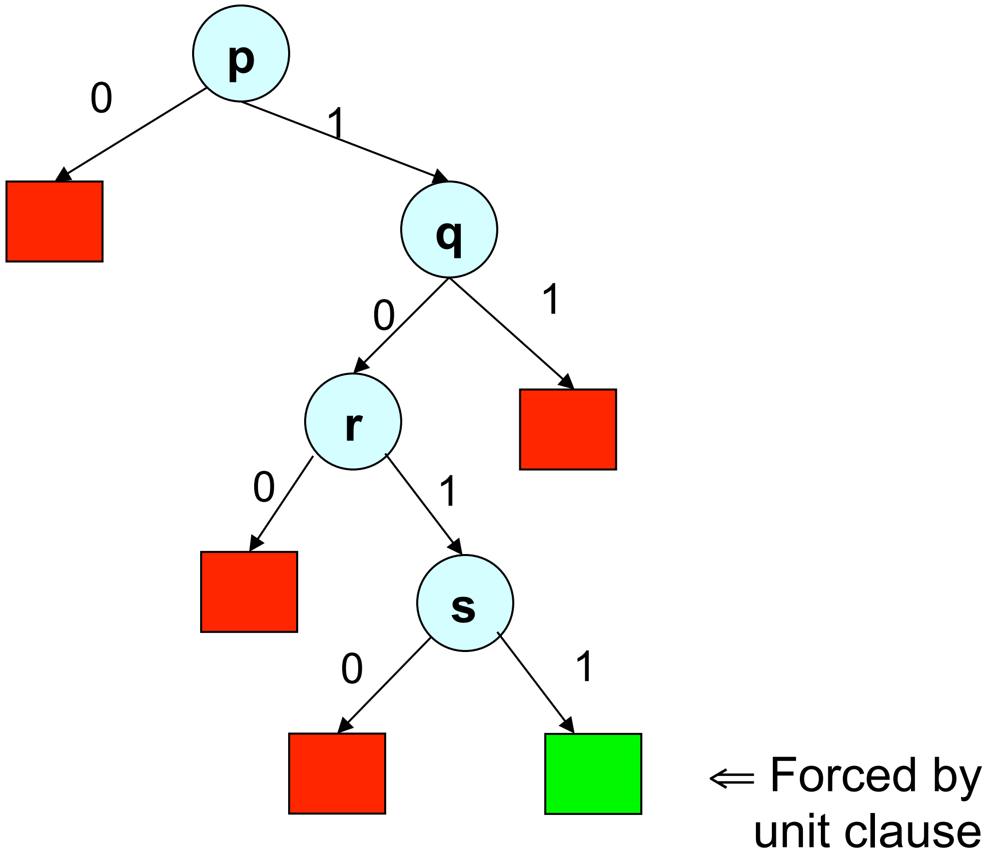
Example, courtesy Leonardo de Moura

Basic DPLL(t) Procedure

Green means "crossed out"

p = 3 < x
q = x < 0
r = x < y
s = y < 0

p
q v r
s v ~r



Example, courtesy Leonardo de Moura