Propositional Logic, DPLL, and Proof Theory

CIS700 — Fall 2022

Kris Micinski

Why talk about logics?

- Logics help us understand and characterize **expressivity limits** of systems
- Logics classify **sets of expressible statements**
  - Sometimes *syntactically* (i.e., sets of formulas you can write)
  - Sometimes *semantically* (a set of interpretations, not necessarily syntactic)
- E.g., **propositional logic**
  - *Atomic* formulas (“Rainy”, “IsSunday”) with boolean interpretation
  - Boolean connectives (and/$\wedge$, or/$\vee$, implies/$\Longrightarrow$, not/$\neg$, …)
- First-order logic allows quantifiers ($\forall x.\ \exists y.\ y \leq x$) — more expressive

Goals for logic segment of this course

- Get an intuition for modern tools for implementing logic at practical scales
  - Will cover several logics: propositional, existential fixed-point logic, first-order logic, higher-order logic, temporal logic, access control logics, …
- Want an intuition: when should I use some logic X to reason about thing Y
  - E.g., use temporal logic to say “user is authorized to access password file after their password has been verified.”
- Will talk about **tools** / libraries to implement these logics

Propositional Logic

- The following are propositional formulas:
  - A, $A \wedge B$, $A \wedge B \implies C$, $A \vee B \implies A$, …
- How do we interpret formulas?
  - This question has a few different answers depending on your outlook on the foundations of mathematics
    - **Classical** — Formulas are “true” or “false,” which we define via an extensionally-specified boolean interpretation
    - **Constructive / Intuitionistic** — “True” formulas are those which have syntactic proofs

Classical Interpretations

- Need a **truth assignment** Γ : Var $\rightarrow$ {True, False} to each variable
- From that, build a truth assignment for formulas, Truth(Φ), depending on Φ:
  - If A is an atom, A is True iff Γ(A) = True, and False otherwise
  - $\phi \wedge \psi$ is True iff Truth(ϕ) = True and Truth(ψ) = True
  - $\phi \vee \psi$ is true iff Truth(ϕ) = True or Truth(ψ) = True
  - $\neg \phi$ is true iff Truth(ϕ) = False (Excluded middle!)
  - $\phi \Longrightarrow \psi$ is true when $\neg \phi \vee \psi$

Classical Binary Connectives

- One possible unary connective (negation)
- Eight possible binary connectives
  - Possible to encode some operators, only need a basis

Symbol, name	Truth table	Venn diagram
Zeroary connectives (constants)		
⊤ Truth/tautology	1	
⊥ Falsity/contradiction	0	
Unary connectives		
	P = 0 1	
Proposition P	0 1	
¬ Negation	1 0	
Binary connectives		
	P = 0 1	
	Q = 0 1 0 1	
Proposition P	0 0 1 1	
Proposition Q	0 1 0 1	
∧ Conjunction	0 0 0 1	
↑ Alternative denial	1 1 1 0	
∨ Disjunction	0 1 1 1	
↓ Joint denial	1 0 0 0	
→ Material conditional	1 1 0 1	
↮ Exclusive or	0 1 1 0	
↔ Biconditional	1 0 0 1	
← Converse implication	1 0 1 1	
More information		

(Wikipedia)

Classical Interpretations

- Some statements are *conditionally true* based on Γ
- **(Definition)** These formulas are called **satisfiable**, as it is possible to find a Γ that solves them. We write this in math as $\Gamma \models \phi$
- Examples:
  - A, true if {A ↦ True}, false if {A ↦ False}
  - $A \wedge B$, true if {A ↦ True, B ↦ True}
  - $C \Longrightarrow A \wedge B$, true if {A ↦False,B↦False,C↦False}/{A ↦True,B ↦True,C↦True};
    - but not when Γ = {A ↦ True, B ↦ False, C ↦ True}

```
(define (formula? ϕ)
  (match ϕ
    [(? symbol? s) #t]
    [`(∧ ,(? formula? ϕ) ,(? formula? ψ)) #t]
    [`(∨ ,(? formula? ϕ) ,(? formula? ψ)) #t]
    [`(⇒ ,(? formula? ϕ) ,(? formula? ψ)) #t]
    [`(¬ ,(? formula? ϕ)) #t]
    [_ #f]))

;; truth :: formula * [ Variable -> bool ] -> bool
(define (truth ϕ ρ)
  (match ϕ
    [(? symbol? s) (hash-ref ρ s)]
    [`(∧ ,(? formula? ϕ) ,(? formula? ψ)) (and (truth ϕ ρ) (truth ψ ρ))]
    [`(∨ ,(? formula? ϕ) ,(? formula? ψ)) (or (truth ϕ ρ) (truth ψ ρ))]
    [`(⇒ ,(? formula? ϕ) ,(? formula? ψ)) (or (not (truth ϕ ρ)) (truth ψ ρ))]
    [`(¬ ,(? formula? ϕ)) (not (truth ϕ ρ))]
    [_ (error "bad formula")]))
```

```
(define (variables ϕ)
  (match ϕ
    [(? symbol? s) (set s)]
    [`(∧ ,(? formula? ϕ) ,(? formula? ψ)) (set-union (variables ϕ) (variables ψ))]
    [`(∨ ,(? formula? ϕ) ,(? formula? ψ)) (set-union (variables ϕ) (variables ψ))]
    [`(⇒ ,(? formula? ϕ) ,(? formula? ψ)) (set-union (variables ϕ) (variables ψ))]
    [`(¬ ,(? formula? ϕ)) (variables ϕ)]))

(define (environments ϕ)
  ;; build all possible configurations of variables
  (define (h vars)
    (match vars
      ['() (list (hash))]
      [`(,x ,vars ...)
       ;; recursive call, build list of all possible
       ;; environments using rest of vars except x
       (define all-environments (h vars))
       ;; now for each of those add x as both true/false
       (append (map (lambda (env) (hash-set env x #t)) all-environments)
               (map (lambda (env) (hash-set env x #f)) all-environments))]))
  (h (set->list (variables ϕ))))
```

Satisfiability Checking

- Given a formula ϕ, is there some Γ wherein ϕ holds?
- For propositional logic, this this is **decidable** via an algorithm
  - Generate all possible environments—check if any of them works!

```
(define (check-sat ϕ)
  (define (h environments)
    (match environments
      ['() #f] ;; no more environments to check
      [`(,ρ ,next-ρs ...) ;; check ρ, return if it works
       (if (truth ϕ ρ) ρ (h next-ρs))]))
  (h (environments ϕ)))
```

Tautologies

- By contrast, some formulas are **always true** *no matter what* Γ is
  - $A \wedge B \implies A$
  - $B \wedge C \implies C \wedge B$
  - $A \vee B \vee \neg B$
- **(Definition)** These formulas, which necessarily hold regardless of valuation assigned by Γ, are called **tautologies**
- We often use the syntax $\models \phi$ to say "ϕ is valid"

Decidability of Propositional Logic

- Question: is it possible to write a program which determines if an arbitrary formula in propositional logic is true?
- **Yes; trivially!** Answer: try all possible interpretations!
- This is called *validity checking* and is **dual** to satisfiability checking

```
(define (valid? ϕ)
  (andmap (lambda (ρ) (truth ϕ ρ)) (environments ϕ)))
```

Satisfiability vs. Validity

- How is it dual?
  - To check validity of ϕ (i.e., ⊨ ϕ), it suffices to check the **satisfiability** of ¬ϕ
- Right now this seems useless—but this holds in more powerful logics too

```
(define (valid-again? ϕ) (not (check-sat `(¬ ,ϕ))))
```

Model Theory

- When an interpretation satisfies a formula, we say the environment Γ is a model for the formula ϕ, $\Gamma \models \phi$
- Field of **model theory** studies logic from the perspective of its models
  - We described a model-theoretic interpretation of truth
  - Model theory asks: what properties are required of models? What is the structure of models? Etc…
- Propositional logic's model theory is, essentially, **truth tables**

P	Q	R	$P \to Q$	$Q \to R$	$(P \to Q) \wedge (Q \to R)$
T	T	T	T	T	T
T	T	F	T	F	F
T	F	T	F	T	F
T	F	F	F	T	F
F	T	T	T	T	T
F	T	F	T	F	F
F	F	T	T	T	T
F	F	F	T	T	T

□

More Intuition; Complexity?

- It is always possible to check the validity / satisfiability of a formula by **constructing the truth table for it**
- Unfortunately, this may get very thorny in practice! Can take a long time!
- **Algorithmic complexity** (space/time) of truth tables is ***exponential***
  - This is the worst-case scenario (beyond undecidability!)
  - Truth table approach never used in practice—proof search preferred instead!

Satisfiability Checking and SAT

- SAT asks the question: given this boolean formula with N variables, is it satisfiable?
  - 3-SAT is **NP-Hard**, but *practical solutions exist*
  - Modern SAT solvers scale to formulas with **millions** of variables
  - http://www.satcompetition.org/
- Lots of SAT variants: MAXSAT, LSAT, Horn-SAT, …
  - Generally complex, but decidable problems requiring special solvers

Why does any of this matter?

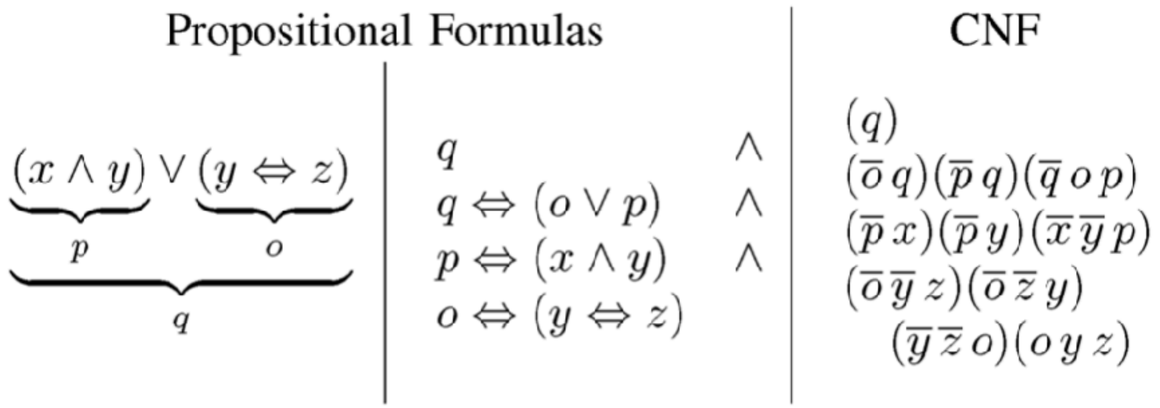
- Propositional logic is a relatively weak, but **decidable** logic—we can always reason about it completely precisely and (in finite time) discern which statements are true and which are false
- Forms the basis for modern EDA algorithms
- **Next week**, “Boolean Satisfiability Solvers and Their Applications in Model Checking” https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7225110

Clausal Normal Forms

- SAT solvers pound through formulas with incredible speed using state-of-the-art top-down search approaches + learning
- Input is **clausal normal form**: conjunction of disjunctions—possible to translate any propositional formula into CNF.

Tseitin's Transformation

- Encodings such as Tseitin's transformation introduce extra variables to flatten a formula into CNF—allows a "flat" input format to solver



"Boolean Satisfiability Solvers and Their Applications in Model Checking" Vizel et. Al

DPLL

- https://kmicinski.com/algorithms/sat/2012/09/22/efficient-sat-solving/
- Basis for modern SAT solving. Basically: guess one variable's value, look for *forced* implications—if you hit a dead end, backtrack
- Modern versions of DPLL augment the backtracking step with *clause learning*—before backtracking, generate a new constraint that forbids you from going off into this part of the search space again

DPLL Example

- A ∨ B ∨ ¬ C, C ∨ D, A ∨ ¬ B
- Pick A = False:
  - (UnitProp, Last clause) ¬ B must be true, thus B must be false
    - (UnitProp, First clause) A/B false, so ¬C must be true; C false
      - (UnitProp, Second clause) Now, D must be true!
        - (No more choices) Our answer is A, B, C = False, D = True

```
Algorithm DPLL
    Input: A set of clauses Φ.
    Output: A truth value indicating whether Φ is satisfiable.
```

```
function DPLL(Φ)
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    if Φ is empty then
        return true;
    if Φ contains an empty clause then
        return false;
    l ← choose-literal(Φ);
    return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {not(l)});
```

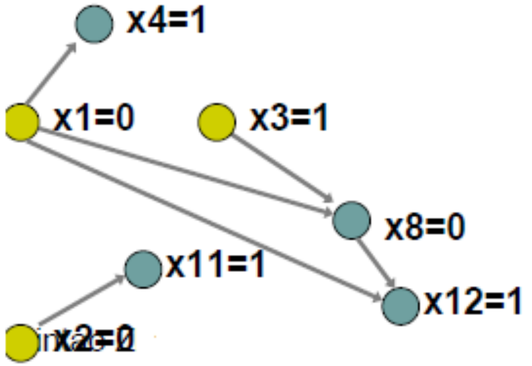
(Wikipedia, DPLL)

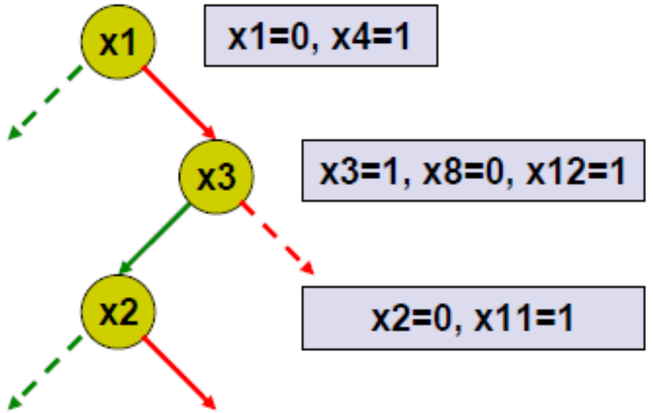
Clause Learning

- Basic idea: when you get to a conflict, learn from your lesson. You don’t want to explore this same junk part of the space again!
- While applying unit propagation, build *constraint graph*—upon conflict, identify the **cut** in the graph that leads to conflict
- Then, learn a synthetic clause that forbids making choices that led you to the conflict

Step 7

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

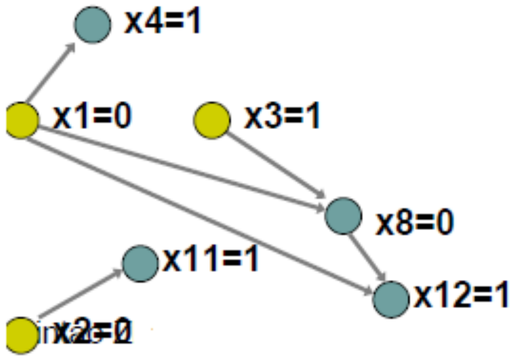


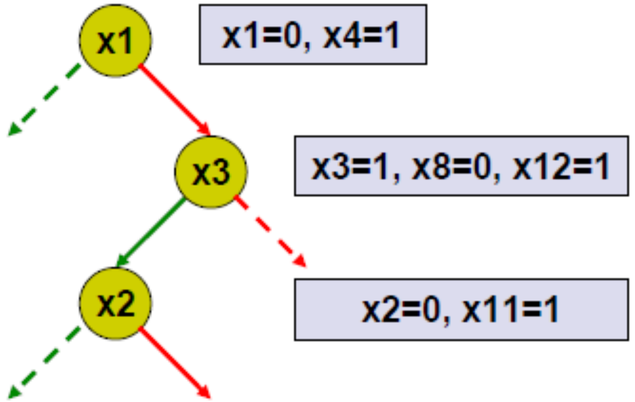


(Wikipedia, Conflict-Directed Clause Learning)

Step 8

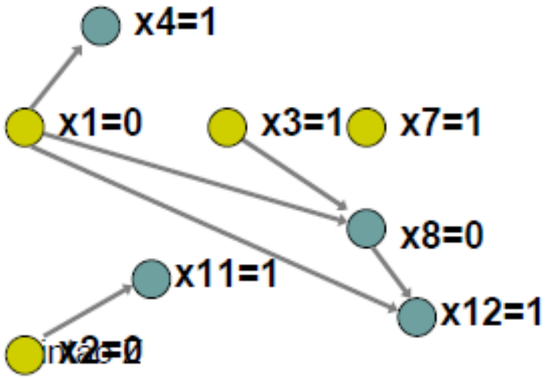
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

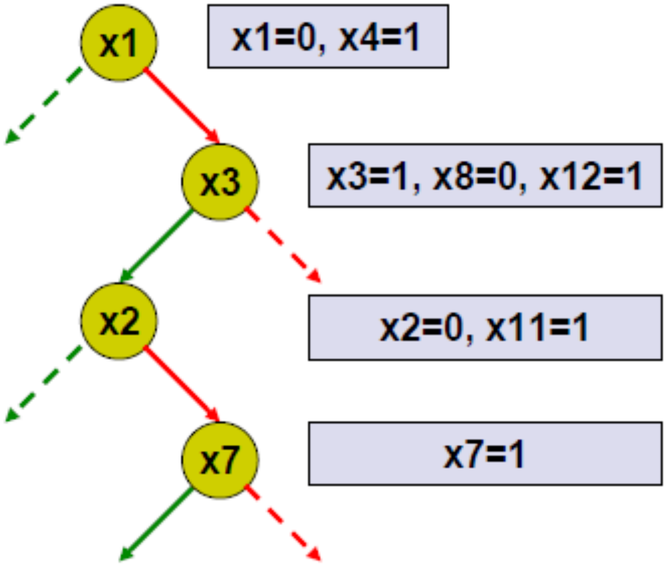




Step 9

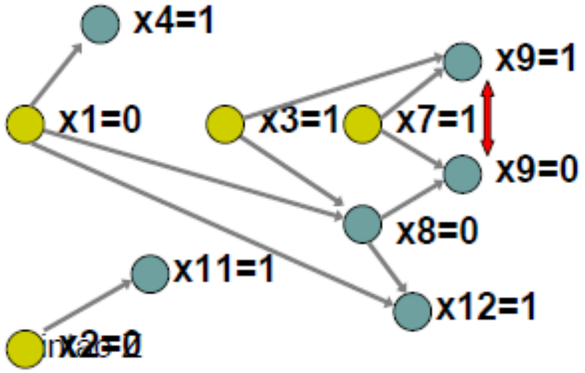
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

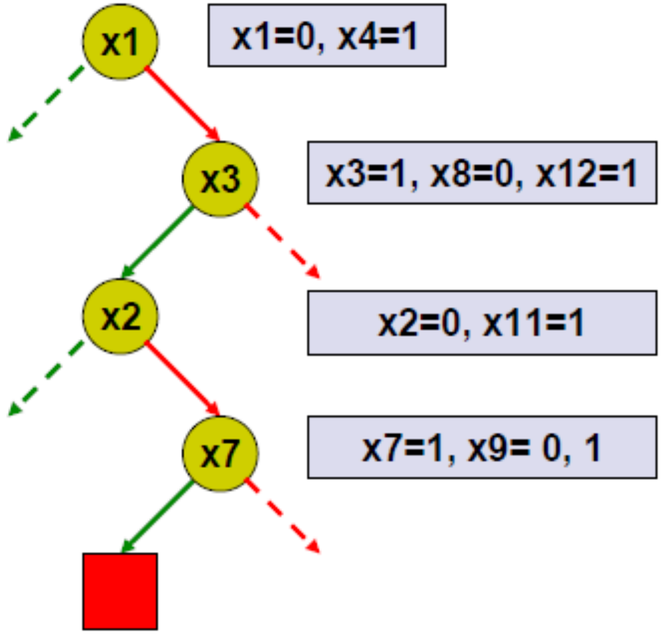




x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

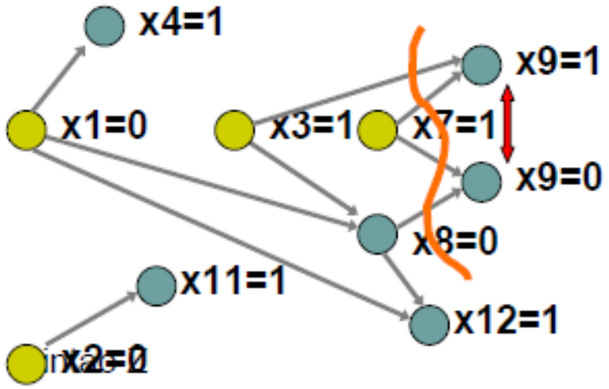
Step 10

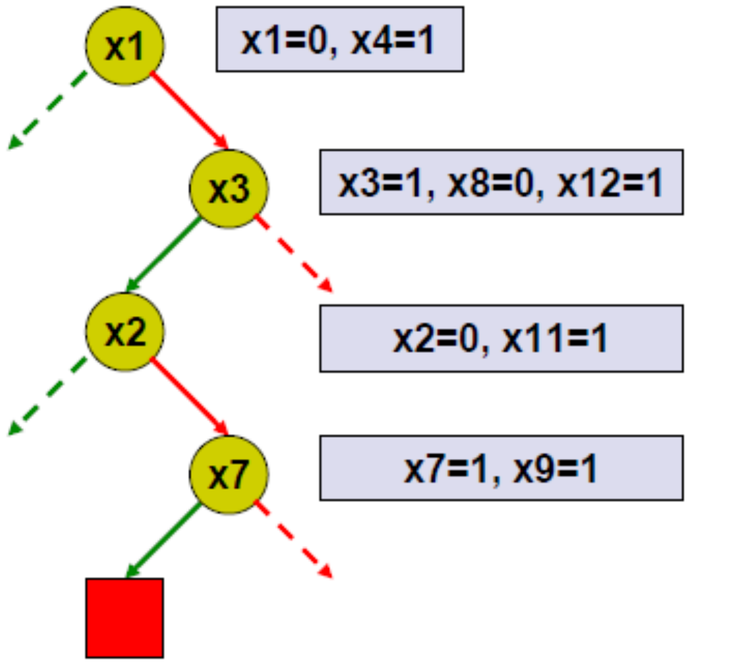




x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

Step 11

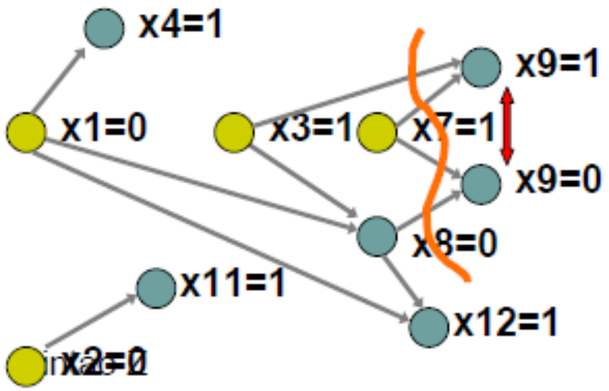


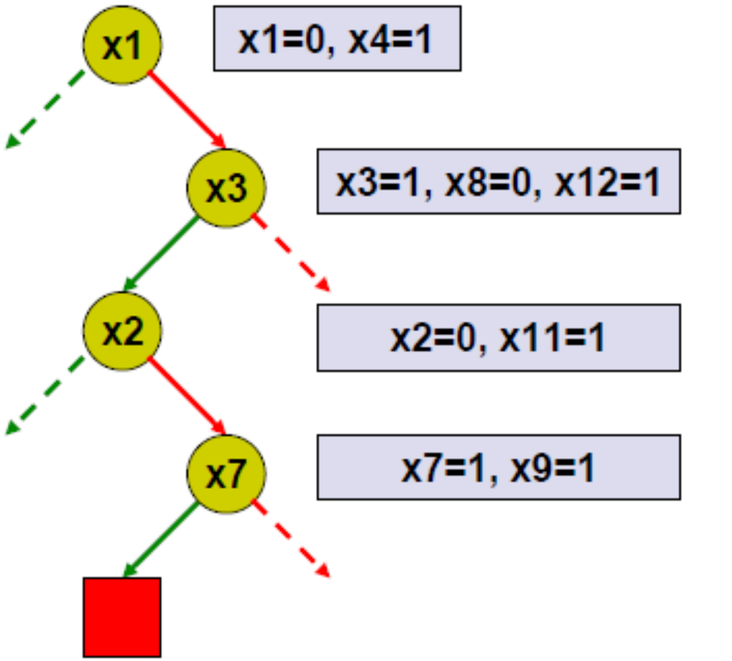


$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow$ conflict

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

Step 11





x3=1∧x7=1∧x8=0 → conflict

If a implies b, then b' implies a'

Step 12

x3=1∧x7=1∧x8=0 → conflict

Not conflict → (x3=1∧x7=1∧x8=0)'

true → (x3=1∧x7=1∧x8=0)'

(x3=1∧x7=1∧x8=0)'

(x3' + x7' + x8)

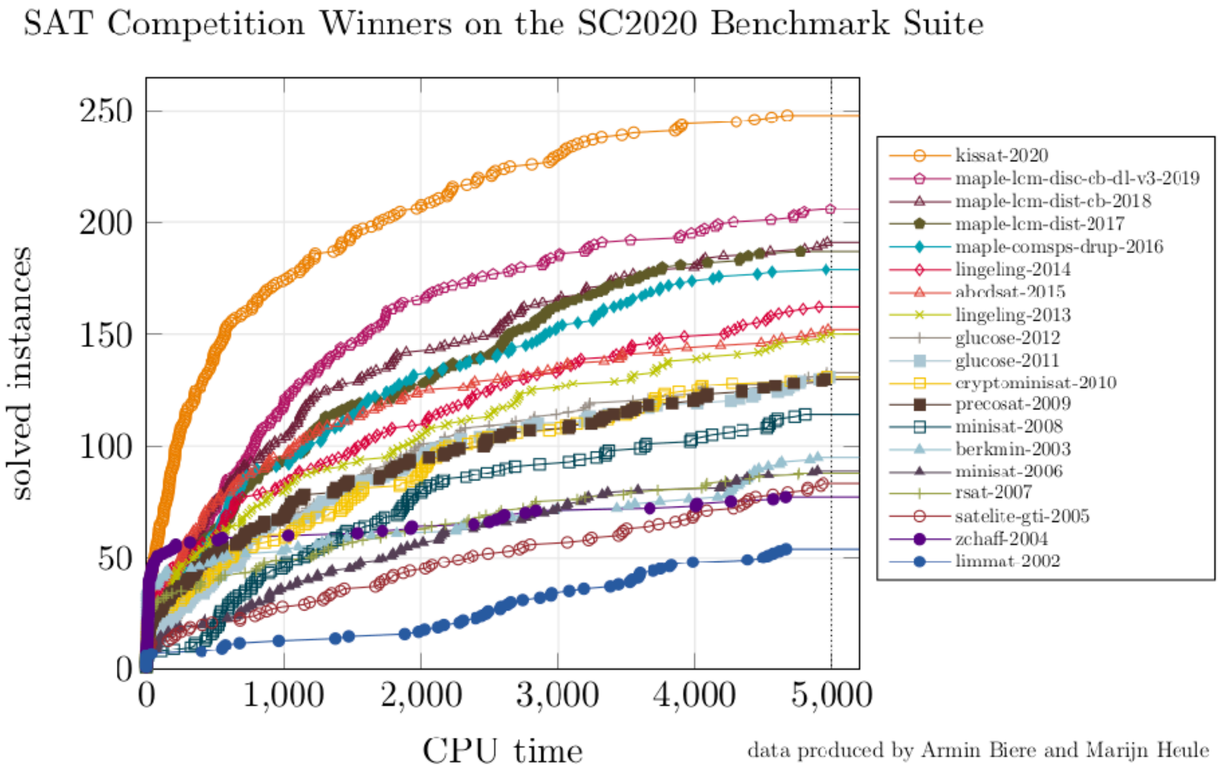
Backtracking

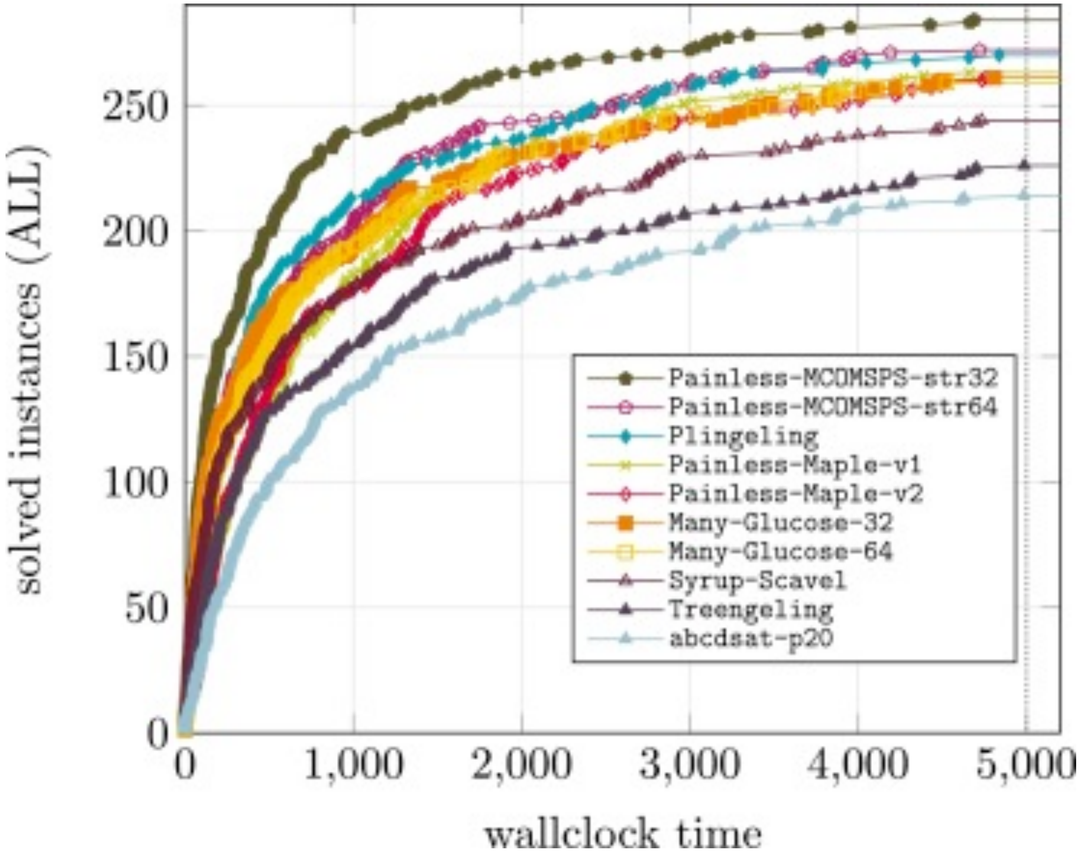
- Once you find a conflict there you can backtrack arbitrarily-far back in history—e.g., to the beginning or another random part of the space
  - In practice restarts do help with some classes of formulae
- Conflict-directed learning + two-watched literal + nonchronological backtracking forms backbone of fast SAT solvers

The Two-Watched-Literals Approach

- http://haz-tech.blogspot.com/2010/08/whos-watching-watch-literals.html?m=1 slides: http://cse.unl.edu/~choueiry/S17-235H/files/SATslides07.pdf
- Need *unit propagation* to be **extremely** fast for efficient SAT
  - Instead of tracking clauses yet-to-be-satisfied, associate with each literal a list of clauses such that the literal is one of two "watch literals."
    - Similar to query planning: do efficient selection via indexing!
  - When assigning A, need to inspect only clauses watching ¬A
  - See above slides for examples…
  - **Very fast! Core data structure used for scalability in practice**

SAT is very scalable in practice!





SAT is relatively weak; does it matter?

- SAT solvers scale to clauses with millions of variables, but:
  - (a) the million variables are in CNF form, and the relationship to high-level formulas that would be *useful* is via an encoding
  - (b) encoding causes blowup in algorithmic complexity (i.e., the millions of variables is over counting)
  - (c) Hard / algorithmically-complex to represent complex data structures in SAT
- Common use cases are things like formulas of 64-length bit-vectors and formulas encoding logical gates in EDA applications
- 10million variables is 156kbytes—so essentially formulas whose solutions are in the kbps range wrt information expressed

Beyond SAT

- We will not focus strongly on the details of SAT solvers
- A variety of techniques extend SAT in various ways with either..
  - Recursion — Datalog (Horn-SAT), Existential Fixed-Point Logic/ Constrained-Horn Clauses
  - Quantification — (over elements) First-order logic, (over types / logical propositions / sorts / …) higher-order logic
- We discuss SAT mainly to give a common baseline for notation

Law of Excluded Middle (LEM)

- Classical logic admits the law of the excluded middle: $\neg\neg A \implies A$
  - Sometimes “proof by contradiction.”
  - However, intuitionism does not disallow all **uses** of contradiction, *just* $\neg\neg A \implies A$—in a constructive setting a proof is a function you can use to give you a proof of anything you want.

Constructivity

- We discussed the **classical** setting so far—*everything is either true or false*. This is why we regard truth as function with range {True, False}
- Another mathematical perspective is *constructivism*
  - A statement is true exactly when there exists some (syntactic) proof of its truth
- We have not yet discussed how proofs are explicitly represented

Proof Theory

- The field of **proof theory** asks how we can *syntactically* characterize the truth of formulas by asking which have (symbolic) proofs
- DPLL is an algorithm, but proof theory is a completely different (symbolic) approach to establishing proof of formulas
- Various systems for doing this: **natural deduction** and **sequent calculus**

Natural Deduction

- Gives **schemas** for proofs; User writes natural deduction proofs which instantiate natural deduction *rules*. Each rule has a set of **antecedents** (assumptions, above the line) and a single **consequent** (conclusion; below the line).

$$\dfrac{\dfrac{A \qquad B}{A \wedge B} \qquad \dfrac{A \qquad C}{A \wedge C}}{(A \wedge B) \wedge (A \wedge C)}$$

This is a **proof** in natural deduction of (A ∧ B) ∧ (A ∧ C)

(From Lean's "Logic and Proof," Ch 3)

https://leanprover.github.io/logic_and_proof/natural_deduction_for_propositional_logic.html

Implication:

$$\dfrac{\dfrac{}{A}\,1}{\vdots}\;\dfrac{B}{A \to B}\,1\ \to\!\mathrm{I}$$

$$\dfrac{A \to B \qquad A}{B}\ \to\!\mathrm{E}$$

Conjunction:

$$\dfrac{A \qquad B}{A \wedge B}\ \wedge\mathrm{I} \qquad \dfrac{A \wedge B}{A}\ \wedge\mathrm{E_l} \qquad \dfrac{A \wedge B}{B}\ \wedge\mathrm{E_r}$$

Truth and falsity:

$$\dfrac{\bot}{A}\ \bot\mathrm{E} \qquad \dfrac{}{\top}\ \top\mathrm{I}$$

Disjunction:

$$\dfrac{A}{A \vee B}\ \vee\mathrm{I_l} \qquad \dfrac{B}{A \vee B}\ \vee\mathrm{I_r}$$

$$\dfrac{A \vee B \qquad \begin{array}{c}\dfrac{}{A}\,1\\ \vdots\\ C\end{array} \qquad \begin{array}{c}\dfrac{}{B}\,1\\ \vdots\\ C\end{array}}{C}\ 1\ \vee\mathrm{E}$$

Negation:

$$\dfrac{\dfrac{}{A}\,1}{\vdots}\;\dfrac{\bot}{\neg A}\,1\ \neg\mathrm{I}$$

$$\dfrac{\neg A \qquad A}{\bot}\ \neg\mathrm{E}$$

Reductio ad absurdum (proof by contradiction):

$$\dfrac{\dfrac{}{\neg A}\,1}{\vdots}\;\dfrac{\bot}{A}\,1\ \mathrm{RAA}$$

Key definitions

- Logical system / set of rules said to be **sound** if everything it proves is “true.”
- A logical system is said to be **consistent** if it is impossible to prove “false.”
- When we say a logic / procedure / … is **complete** we mean (roughly) “every provable thing is true.”
- Natural deduction is sound and complete for propositional logic—it is always possible to find a natural deduction proof for any true statement, and every natural deduction proof is of a true statement (under its assumptions).

Why should we care?

- Proof theory focuses on the *explicit materialization* of proofs—this allows us to construct certificates of correctness for code we write
  - Lets us build correct and secure systems
- Lots of practical, real-world security problems can be phrased in terms of **software correctness**
- Can use automated **proof assistants** (based on some higher-order / dependent type theory) to develop and check our proofs

CertiCrypt — Formally-verified cryptography in Coq

```
313     let EA   := add_decl Eenc A A_params (refl_equal true) A_body A_ret in
314      add_decl EA A' A'_params (refl_equal true) A'_body A'_ret.
315
316   (** Environment for the DDH and ES adversaries *)
317   Definition EBD :=
318    let ED := add_decl E D D_params (refl_equal true) D_body D_res in
319     add_decl ED B B_params (refl_equal true) B_body B_res.
320
321   (** The set of oracles that can be called by [A] and [A'] (there are none) *)
322   Definition PrOrcl := PrSet.empty.
323
324   (** Private procedures, not accessible to the adversary *)
325   Definition PrPriv := PrSet.add (BProc.mkP B) (PrSet.singleton (BProc.mkP D)).
326
327   (** The adversary is well-formed in [E], i.e. it only reads or writes
328      variables it has access to, and only calls oracles and its own procedures *)
329   Hypothesis A_wf  : WFAdv PrOrcl PrPriv Gadv Gcomm E A.
330   Hypothesis A'_wf : WFAdv PrOrcl PrPriv Gadv Gcomm E A'.
331
332   (** The adversary runs in PPT *)
333   Hypothesis A_PPT  : forall E', Eq_adv_decl PrOrcl PrPriv E E' -> PPT_proc E' A.
334   Hypothesis A'_PPT : forall E', Eq_adv_decl PrOrcl PrPriv E E' -> PPT_proc E' A'.
335
336   (** The adversary is lossless (i.e. it always terminates) *)
337   Hypothesis A_lossless  : forall E, lossless E A_body.
338   Hypothesis A'_lossless : forall E, lossless E A'_body.
339
340   Lemma EqAD : Eq_adv_decl PrOrcl PrPriv E EBD.
341   Proof.
342    unfold Eq_adv_decl, proc_params, proc_body, proc_res; intros.
343    generalize (BProc.eqb_spec (BProc.mkP A) (BProc.mkP f)).
344    destruct (BProc.eqb (BProc.mkP A) (BProc.mkP f)); intros.
```

Summary

- Propositional logic is decidable and useful, but lacks expressivity of more complete logics.
  - E.g., propositional logic can’t express linear arithmetic without specific encodings (e.g., bit vectors)
  - Propositional logics don’t allow quantifiers
- Natural deduction explicitly materializes proofs for propositional formulas
  - Key idea: do you **materialize** a proof or just *trust* a solver?

Tools

- minisat http://logicrunch.it.uu.se:4096/~wv/minisat/
- Z3 https://microsoft.github.io/z3guide/playground/Freeform%20Editing/