H250: Honors Colloquium – Introduction to Computation

Satisfiability Checking

Marius Minea
marius@cs.umass.edu

Review: Satisfiable formulas

A formula is *satisfiable*
if at least one *truth assignment* for its variables makes it *true*.

i.e., there is an *interpretation* that satisfies it

there is a line in the *truth table* with result *true*

Is the following formula satisfiable?

$$(\neg p \wedge \neg q) \wedge (\neg r \vee \neg(p \vee \neg(\neg r \rightarrow q)))$$

Maybe not at first sight, but can do it *systematically*

Option 1: try all cases

Formula: $(\neg p \wedge \neg q) \wedge (\neg r \vee \neg(p \vee \neg(\neg r \rightarrow q)))$

$p = \mathsf{T}, q = \mathsf{T}, r = \mathsf{T}$: $(\mathsf{F} \wedge \mathsf{F}) \wedge (\mathsf{F} \vee \neg(\mathsf{T} \vee \neg(\mathsf{F} \rightarrow T)))$
    is *false*
$p = \mathsf{T}, q = \mathsf{T}, r = \mathsf{F}$: $(\mathsf{F} \wedge \mathsf{F}) \wedge (\mathsf{T} \vee \neg(\mathsf{T} \vee \neg(\mathsf{T} \rightarrow T)))$
    is *false*

...
in effect, build *truth table* until
  a *true* value is found
  or we exhaust all options

Option 2: case split and simplify

Formula: $f = (\neg p \wedge \neg q) \wedge (\neg r \vee \neg(p \vee \neg(\neg r \rightarrow q)))$

Let's try $p = \mathsf{T}$.

$$\begin{aligned} f\,\big|_{p=\mathsf{T}} &= (\mathsf{F} \wedge \neg q) \wedge (\neg r \vee \neg(\mathsf{T} \vee \neg(\neg r \rightarrow q))) \\ &= \mathsf{F} \wedge (\neg r \vee \neg(\mathsf{T} \vee \neg(\neg r \rightarrow q))) \\ &= \mathsf{F} \end{aligned}$$

not successful, but saves us from trying all subcases for q and r

Option 2: case split and simplify

Formula: $f = (\neg p \wedge \neg q) \wedge (\neg r \vee \neg(p \vee \neg(\neg r \rightarrow q)))$

Let's try $p = \mathsf{T}$.

$$
\begin{aligned}
f\,\big|_{p=\mathsf{T}} &= (\mathsf{F} \wedge \neg q) \wedge (\neg r \vee \neg(\mathsf{T} \vee \neg(\neg r \rightarrow q))) \\
&= \mathsf{F} \wedge (\neg r \vee \neg(\mathsf{T} \vee \neg(\neg r \rightarrow q))) \\
&= \mathsf{F}
\end{aligned}
$$

not successful, but saves us from trying all subcases for q and r

We try $p = \mathsf{F}$.

$$
\begin{aligned}
f\,\big|_{p=\mathsf{F}} &= (\mathsf{T} \wedge \neg q) \wedge (\neg r \vee \neg(\mathsf{F} \vee \neg(\neg r \rightarrow q))) \\
&= \neg q \wedge (\neg r \vee \neg\neg(\neg r \rightarrow q)) \\
&= \neg q \wedge (\neg r \vee r \vee q) \\
&= \neg q
\end{aligned}
$$

and the formula is true for $p = q = \mathsf{F}$, thus satisfiable

better option, but can still make poor choices, redo computations

Why is SAT checking relevant?

Theory: first problem shown to be NP-complete

Logic design and circuit verification:
functions for original and optimized circuit
equivalence means $\neg(f_{orig} \leftrightarrow f_{opt})$ NOT satisfiable

Program analysis and verification
can we traverse successive `if` statements on a certain path?

Constraint satisfaction:

Scheduling
all classes scheduled, no two conflicting, at most 8 hours/day

Planning
actions occur in certain order, one outcome conditions another

Bioinformatics: making inferences from genotype data

also familar puzzles: n-queens, Sudoku, etc.

A motivating puzzle

Can you find a *sequence of 8 bits* b_1, b_2, ..., b_8 that has
no three equally-spaced 0s
no three equally-spaced 1s

if interested: *van der Waerden numbers*

For instance, 00101101 is not good: has 0 at positions 1, 4, and 7

positions 1, 2, 3 cannot be all zeroes, nor all ones

$$(b_1 \vee b_2 \vee b_3) \wedge (\neg b_1 \vee \neg b_2 \vee \neg b_3)$$

same for positions (2, 3, 4), ..., (6, 7, 8) (spacing 1)
also for positions (1, 3, 5), ..., (4, 6, 8) (spacing 2)
and for (1, 4, 7) and (2, 5, 8) (spacing 3)

$$\ldots \wedge (b_1 \vee b_4 \vee b_7) \wedge (\neg b_1 \vee \neg b_4 \vee \neg b_7)$$
$$\wedge (b_2 \vee b_5 \vee b_8) \wedge (\neg b_2 \vee \neg b_5 \vee \neg b_8)$$

We have a formula structured as a *conjunction* of constraints
common in constraint satisfaction problems

Review: Conjunctive Normal Form

A formula is in conjunctive normal form, if it is
a *conjunction* $\wedge$ of *clauses*, where
a clause is a *disjunction* $\vee$ of literals, and
a literal is a *propositional variable* p or its *negation* $\neg p$

In other words, the formula is structured as an **AND** of **OR**s

$$\begin{aligned} & (a \vee \neg b \vee \neg d) \\ \wedge\ & (\neg a \vee \neg b) \\ \wedge\ & (\neg a \vee c \vee \neg d) \\ \wedge\ & (\neg a \vee b \vee c) \end{aligned}$$

Can you see a way to make the formula true?

Advantages of CNF

$$\begin{aligned} & (a \vee \neg b \vee \neg d) \\ \wedge\ & (\neg a \vee \neg b) \\ \wedge\ & (\neg a \vee c \vee \neg d) \\ \wedge\ & (\neg a \vee b \vee c) \end{aligned}$$

Having a formula in a regular form is of advantage:

ease of representation: list of lists of literals

ease of processing: few cases to handle

Constraint compositions are often close to conjunctive normal form
we've seen how to transform a formula to CNF
and avoid exponential blowup (Tseitin transform)

Let's find some rules that make checking satisfiability easier, allowing us to *simplify* the problem.

Rules for satisfiability: Unit clause

R_{UNIT}: A single literal (*unit clause*) must be assigned *true*.
(one-literal rule)

$$\begin{array}{lll} & & a \\ \text{in} & \wedge & (\neg a \vee b \vee c) \\ & \wedge & (\neg a \vee \neg b \vee \neg c) \end{array}$$

a must be taken T

$$\begin{array}{lll} & & (a \vee b) \\ \text{in} & \wedge & \neg b \\ & \wedge & (\neg a \vee \neg b \vee c) \end{array}$$

b must be taken F

otherwise the formula is false

Rules for satisfiability: Unit clause

$\mathrm{R}_{\mathrm{UNIT}}$: A single literal (*unit clause*) must be assigned *true*.
(one-literal rule)

		a	
in	$\wedge$	$(\neg a \vee b \vee c)$	a must be taken T
	$\wedge$	$(\neg a \vee \neg b \vee \neg c)$	

		$(a \vee b)$	
in	$\wedge$	$\neg b$	b must be taken F
	$\wedge$	$(\neg a \vee \neg b \vee c)$	

otherwise the formula is false

If we find any unit clauses, the values imposed for those literals allow us to simplify the formula further.

Rules for satisfiability: Boolean constraint propagation

R_{BCP1}: If a literal L is T, *delete clauses containing* L
they are true (as desired), no need to consider further

R_{BCP2}: If a literal L is F, *delete it* from all clauses
false can't help make clause *true*

Previous examples simplify:

$$
\begin{array}{ll}
 & a \\
\wedge & (\cancel{\neg a} \vee b \vee c) \\
\wedge & (\cancel{\neg a} \vee \neg b \vee \neg c)
\end{array}
\stackrel{a=\mathrm{T}}{\rightarrow}
\begin{array}{ll}
 & (b \vee c) \\
\wedge & (\neg b \vee \neg c)
\end{array}
$$

$$
\begin{array}{ll}
 & (a \vee \cancel{b}) \\
\wedge & \neg b \\
\wedge & (\neg a \vee \neg b \vee c)\checkmark
\end{array}
\stackrel{b=\mathrm{F}}{\rightarrow} a
$$

and from here $a = T$, formula is satisfiable

Rules for satisfiability: Stopping

R_{STOP}: If there are *no more clauses*, formula is satisfiable with the truth assignment obtained so far

If we have an *empty clause*, formula is not satisfiable
no literals in empty clause to make it true

$$\begin{array}{ll} & (a \vee b)\checkmark \\ \wedge & a\checkmark \\ \wedge & (a \vee \neg b \vee c)\checkmark \end{array} \quad \stackrel{a=\mathrm{T}}{\rightarrow} \quad \mathit{SAT}$$

$$\begin{array}{ll} & b\checkmark \\ \wedge & (\cancel{\neg b} \vee c) \\ \wedge & (\cancel{\neg b} \vee \neg c) \end{array} \quad \stackrel{b=\mathrm{T}}{\rightarrow} \quad \begin{array}{ll} & c\checkmark \\ \wedge & \cancel{\neg c} \end{array} \quad \stackrel{c=\mathrm{T}}{\rightarrow} \quad \emptyset \quad \mathit{UNSAT}$$

$c = \mathrm{T}$ makes $\neg c$ empty clause $\Rightarrow$ not satisfiable

Rules for satisfiability: Stopping

R_{STOP}: If there are *no more clauses*, formula is satisfiable with the truth assignment obtained so far

If we have an *empty clause*, formula is not satisfiable no literals in empty clause to make it true

$$
\begin{array}{ll}
 & (a \lor b)\checkmark \\
\land & a\checkmark \\
\land & (a \lor \neg b \lor c)\checkmark
\end{array}
\quad \stackrel{a=\mathrm{T}}{\to} \quad \mathit{SAT}
$$

$$
\begin{array}{ll}
 & b\checkmark \\
\land & (\cancel{\neg b} \lor c) \\
\land & (\cancel{\neg b} \lor \neg c)
\end{array}
\quad \stackrel{b=\mathrm{T}}{\to} \quad
\begin{array}{ll}
 & c\checkmark \\
\land & \cancel{\neg c}
\end{array}
\quad \stackrel{c=\mathrm{T}}{\to} \quad \emptyset \quad \mathit{UNSAT}
$$

$c = \mathrm{T}$ makes $\neg c$ empty clause $\Rightarrow$ not satisfiable

Both stopping conditions have *empty* as list base case:

empty clause list means SAT: all clauses accounted for ✓

empty clause means UNSAT: no literals to make it true

Rules for satisfiability: Case splitting

What if no more simplifications can be done?

$a \wedge (\neg a \vee b \vee c) \wedge (\neg b \vee \neg c) \quad \stackrel{a=\mathsf{T}}{\rightarrow} \quad (b \vee c) \wedge (\neg b \vee \neg c) \quad ??$

R$_{\mathsf{CASE}}$: Choose a proposition and *split by cases* (try both options):

- with value F
- with value T

Any solution is fine

If *none of the two* cases has solution, formula is *unsatisfiable*

A solution algorithm

We are given

- a list of clauses (the formula)
- the set of already assigned literals (initially empty)

Rules $\mathrm{R}_{\mathrm{UNIT}}$ and $\mathrm{R}_{\mathrm{BCP}}$ *reduce the problem* to a *simpler* one (fewer propositions or fewer/simpler clauses)

Rule $\mathrm{R}_{\mathrm{STOP}}$ tells us we are done (have an answer)

Rule $\mathrm{R}_{\mathrm{CASE}}$ reduces the problem to *two simpler problems* (one less propositional variable)

Reducing the problem to one or more simpler instances of itself
$\Rightarrow$ we have a *recursive solution* (with stopping condition $\mathrm{R}_{\mathrm{STOP}}$)

Writing a pseudocode algorithm

We'll either return the set of literals assigned T (can use to check) or raise an exception Unsat

function solve(clauses: clause list, truelit: lit set)
while clauses contains a unit clause **do**
  (clauses, truelit) = simplify(clauses, truelit) (* R_{UNIT}, R_{BCP}*)

Writing a pseudocode algorithm

We'll either return the set of literals assigned T (can use to check) or raise an exception Unsat

```
function solve(clauses: clause list, truelit: lit set)
while clauses contains a unit clause do
   (clauses, truelit) = simplify(clauses, truelit) (* R_UNIT, R_BCP *)
if clauses = empty list then
   return  truelit    (* R_STOP: SAT, return set of true literals *)
```

Writing a pseudocode algorithm

We'll either return the set of literals assigned T (can use to check) or raise an exception Unsat

function solve(clauses: clause list, truelit: lit set)
while clauses contains a unit clause **do**
  (clauses, truelit) = simplify(clauses, truelit) (* R_{UNIT}, R_{BCP}*)
if clauses = empty list **then**
  return truelit (* R_{STOP}: SAT, return set of true literals *)
if clauses contains empty clause **then**
  raise Unsat (* R_{STOP}: UNSAT *)

Writing a pseudocode algorithm

We'll either return the set of literals assigned T (can use to check)
or raise an exception Unsat

function solve(clauses: clause list, truelit: lit set)
while clauses contains a unit clause **do**
  (clauses, truelit) = simplify(clauses, truelit) (* R_{UNIT}, R_{BCP}*)
if clauses = empty list **then**
  return truelit (* R_{STOP}: SAT, return set of true literals *)
if clauses contains empty clause **then**
  raise Unsat (* R_{STOP}: UNSAT *)
choose a literal p
try solve (clauses, truelit $\cup \{\neg p\}$) (* R_{CASE}: try p=F *)
with Unsat $\rightarrow$ solve (clauses, truelit $\cup\{p\}$) (* try p=T *)

Davis-Putnam-Logemann-Loveland algorithm (1962)
simplified, no check for *pure* literals (just positive/just negated)

Applying Boolean Constraint Propagation

True literals: $\emptyset$

$\neg x_1 \vee x_2$

$x_3 \vee x_4 \vee x_5$

x_1

$x_1 \vee x_7$

$\neg x_1 \vee \neg x_4$

$\neg x_3 \vee x_5$

$\neg x_5 \vee x_6$

Traverse list of clauses, accumulate a new list of filtered clauses, and a set of literals found *true* from unit clauses

Applying Boolean Constraint Propagation

True literals: $\emptyset$

$\Rightarrow \neg x_1 \vee x_2$	$\neg x_1 \vee x_2$
$x_3 \vee x_4 \vee x_5$	
x_1	
$x_1 \vee x_7$	
$\neg x_1 \vee \neg x_4$	
$\neg x_3 \vee x_5$	
$\neg x_5 \vee x_6$	

No true literals yet, can't simplify, just copy

Applying Boolean Constraint Propagation

	True literals:	$\emptyset$
	$\neg x_1 \vee x_2$	$\neg x_1 \vee x_2$
$\Rrightarrow$	$x_3 \vee x_4 \vee x_5$	$x_3 \vee x_4 \vee x_5$
	x_1	
	$x_1 \vee x_7$	
	$\neg x_1 \vee \neg x_4$	
	$\neg x_3 \vee x_5$	
	$\neg x_5 \vee x_6$	

No true literals yet, can't simplify, just copy

Applying Boolean Constraint Propagation

True literals:	$\emptyset$
$\neg x_1 \vee x_2$	$\neg x_1 \vee x_2$
$x_3 \vee x_4 \vee x_5$	$x_3 \vee x_4 \vee x_5$
$\Rightarrow x_1$	x_1
$x_1 \vee x_7$	
$\neg x_1 \vee \neg x_4$	
$\neg x_3 \vee x_5$	
$\neg x_5 \vee x_6$	

First unit clause found.

Restart simplification of new clause list with x_1 set to T

Applying BCP: Eagerly re-simplify with unit clause

True literals:	$\emptyset$	$\{1\}$
$\neg x_1 \vee x_2$	$\neg x_1 \vee x_2$	
$x_3 \vee x_4 \vee x_5$	$x_3 \vee x_4 \vee x_5$	
x_1		
$x_1 \vee x_7$		
$\neg x_1 \vee \neg x_4$		
$\neg x_3 \vee x_5$		
$\neg x_5 \vee x_6$		

Simplify clauses accumulated so far with x_1 set to T

Applying BCP: Eagerly re-simplify with unit clause

True literals:		$\emptyset$	$\{1\}$
$\neg x_1 \vee x_2$	$\Rrightarrow$	$\neg x_1 \vee x_2$	x_2
$x_3 \vee x_4 \vee x_5$		$x_3 \vee x_4 \vee x_5$	
x_1			
$x_1 \vee x_7$			
$\neg x_1 \vee \neg x_4$			
$\neg x_3 \vee x_5$			
$\neg x_5 \vee x_6$			

$\neg x_1$ is filtered from first clause
New unit clause x_2 is found

Applying BCP: continue with two true literals

True literals:	$\emptyset$	$\{1, 2\}$
$\neg x_1 \vee x_2$	$\Rightarrow x_3 \vee x_4 \vee x_5$	$x_3 \vee x_4 \vee x_5$
$x_3 \vee x_4 \vee x_5$		
x_1		
$x_1 \vee x_7$		
$\neg x_1 \vee \neg x_4$		
$\neg x_3 \vee x_5$		
$\neg x_5 \vee x_6$		

Done at this level, next clause not changed

Applying BCP: return to first-level traversal

True literals: $\{1, 2\}$

$\neg x_1 \vee x_2$

$x_3 \vee x_4 \vee x_5$

x_1

$\Rrightarrow x_1 \vee x_7$ ✓

$\neg x_1 \vee \neg x_4$

$\neg x_3 \vee x_5$

$\neg x_5 \vee x_6$

$x_3 \vee x_4 \vee x_5$

Clause is *true*, ignored

Applying BCP: return to first-level traversal

True literals:	$\{1, 2\}$
$\neg x_1 \vee x_2$	$x_3 \vee x_4 \vee x_5$
$x_3 \vee x_4 \vee x_5$	$\neg x_4$
x_1	
$x_1 \vee x_7$ ✓	
$\Rrightarrow$ $\neg x_1 \vee \neg x_4$	
$\neg x_3 \vee x_5$	
$\neg x_5 \vee x_6$	

New unit clause $\neg x_4$ found

Applying BCP: restart simplification

True literals:	$\{1, 2\}$	$\{1, 2, -4\}$
$\neg x_1 \vee x_2$	$\Rrightarrow x_3 \vee x_4 \vee x_5$	$x_3 \vee x_5$
$x_3 \vee x_4 \vee x_5$		
x_1		
$x_1 \vee x_7$		
$\neg x_1 \vee \neg x_4$		
$\neg x_3 \vee x_5$		
$\neg x_5 \vee x_6$		

Only clause in list simplifies to $x_3 \vee x_5$

Applying BCP: continue at first level

True literals: $\{1, 2, -4\}$

$\neg x_1 \vee x_2$

$x_3 \vee x_4 \vee x_5$

x_1

$x_1 \vee x_7$

$\neg x_1 \vee \neg x_4$

$\Rrightarrow$ $\neg x_3 \vee x_5$

$\neg x_5 \vee x_6$

$x_3 \vee x_5$

$\neg x_3 \vee x_5$

Applying BCP: continue at first level

True literals:	$\{1, 2, -4\}$
$\neg x_1 \vee x_2$	$x_3 \vee x_5$
$x_3 \vee x_4 \vee x_5$	$\neg x_3 \vee x_5$
x_1	$\neg x_5 \vee x_6$
$x_1 \vee x_7$	
$\neg x_1 \vee \neg x_4$	
$\neg x_3 \vee x_5$	
$\Rightarrow \neg x_5 \vee x_6$	

Last two clauses copied unchanged.

Three literals have been assigned *true*.
Formula has been simplified to three clauses.

Example: our 8-bit puzzle

Find a *sequence of 8 bits* b_1, b_2, ..., b_8 that has
no three equally-spaced 0s
no three equally-spaced 1s
All clauses are triples (minus denotes negated literal)

```
[[1; 2; 3]; [-1; -2; -3]; [2; 3; 4]; [-2; -3; -4];
 [3; 4; 5]; [-3; -4; -5]; [4; 5; 6]; [-4; -5; -6];
 [5; 6; 7]; [-5; -6; -7]; [6; 7; 8]; [-6; -7; -8];
 [1; 3; 5]; [-1; -3; -5]; [2; 4; 6]; [-2; -4; -6];
 [3; 5; 7]; [-3; -5; -7]; [4; 6; 8]; [-4; -6; -8];
 [1; 4; 7]; [-1; -4; -7]; [2; 5; 8]; [-2; -5; -8]]
```

```
solution: [-8; -7; -4; -3; 1; 2; 5; 6]
```

One solution: 11001100

Example: our 8-bit puzzle

Find a *sequence of 8 bits* b_1, b_2, ..., b_8 that has

no three equally-spaced 0s

no three equally-spaced 1s

All clauses are triples (minus denotes negated literal)

```
[[1; 2; 3]; [-1; -2; -3]; [2; 3; 4]; [-2; -3; -4];
 [3; 4; 5]; [-3; -4; -5]; [4; 5; 6]; [-4; -5; -6];
 [5; 6; 7]; [-5; -6; -7]; [6; 7; 8]; [-6; -7; -8];
 [1; 3; 5]; [-1; -3; -5]; [2; 4; 6]; [-2; -4; -6];
 [3; 5; 7]; [-3; -5; -7]; [4; 6; 8]; [-4; -6; -8];
 [1; 4; 7]; [-1; -4; -7]; [2; 5; 8]; [-2; -5; -8]]
```

```
solution: [-8; -7; -4; -3; 1; 2; 5; 6]
```

One solution: 11001100

How could we find another one?

Complexity of SAT checking

A formula with n propositions has 2^n truth assignments
$\Rightarrow$ *exponential time* trying all

But: a given truth assignment can be checked in *polynomial time* (in formula size): traverse formula once and compute value

In general, *checking* a solution is (much) easier than *finding* it.

NP (nondeterministic polynomial time): class of problems for which a solution ("guessed" or given) can be *verified* in polynomial time.

SAT-checking is the first problem proved *NP-complete* (Cook'71): solving SAT-checking in polynomial time would imply $P = NP$

No such algorithm is known, but huge practical progress in solvers:
million variables, tens of millions of clauses
yearly tool competitions, strong industrial usage

Limitations of basic DPLL

- *Naive decisions*: literal on which to branch chosen arbitrarily

Limitations of basic DPLL

- *Naive decisions*: literal on which to branch chosen arbitrarily
  $\Rightarrow$ Heuristics for better choice

Limitations of basic DPLL

- *Naive decisions*: literal on which to branch chosen arbitrarily
  $\Rightarrow$ Heuristics for better choice
- *Does not learn* from conflicts: a bad partial assignment may be retried multiple times (in different subtrees)

Limitations of basic DPLL

- *Naive decisions*: literal on which to branch chosen arbitrarily
  $\Rightarrow$ Heuristics for better choice
- *Does not learn* from conflicts: a bad partial assignment may be retried multiple times (in different subtrees)
  $\Rightarrow$ *clause learning*: learn new clauses (constraints) that need to be satisfied
  quad e.g., $(A \vee p) \wedge (B \vee \neg p) \rightarrow (A \vee B)$

Limitations of basic DPLL

- *Naive decisions*: literal on which to branch chosen arbitrarily
  $\Rightarrow$ Heuristics for better choice
- *Does not learn* from conflicts: a bad partial assignment may be retried multiple times (in different subtrees)
  $\Rightarrow$ *clause learning*: learn new clauses (constraints) that need to be satisfied
  quad e.g., $(A \vee p) \wedge (B \vee \neg p) \rightarrow (A \vee B)$
- *Backtracks one level*, but root of conflict may be higher up

Limitations of basic DPLL

- *Naive decisions*: literal on which to branch chosen arbitrarily
  $\Rightarrow$ Heuristics for better choice
- *Does not learn* from conflicts: a bad partial assignment may be retried multiple times (in different subtrees)
  $\Rightarrow$ *clause learning*: learn new clauses (constraints) that need to be satisfied
  quad e.g., $(A \vee p) \wedge (B \vee \neg p) \rightarrow (A \vee B)$
- *Backtracks one level*, but root of conflict may be higher up
  $\Rightarrow$*non-chronological backtracking*: several levels up, avoids pointless search

Limitations of basic DPLL

- *Naive decisions*: literal on which to branch chosen arbitrarily
  $\Rightarrow$ Heuristics for better choice
- *Does not learn* from conflicts: a bad partial assignment may be retried multiple times (in different subtrees)
  $\Rightarrow$ *clause learning*: learn new clauses (constraints) that need to be satisfied
  quad e.g., $(A \vee p) \wedge (B \vee \neg p) \rightarrow (A \vee B)$
- *Backtracks one level*, but root of conflict may be higher up
  $\Rightarrow$*non-chronological backtracking*: several levels up, avoids pointless search

Conflict Driven Clause Learning: technique used in family of modern solvers

Rephrasing the Algorithm

Main actions:

- *Decide* (assign a variable)
- *BCP* (Boolean Constraint Propagation)
- *AnalyzeConflict*: determine backtracking

```
dlevel = 0
while NotYetSatisfied do
    Decide(); ++dlevel;
    if BCP() == conflict then
        dlevel = AnalyzeConflict()
        if dlevel < 0 then
            return UNSAT
return sat-assignment
```

Techniques in modern solvers

- *Decision level* for each variable assignment
  if assigned through BCP, same decision level
  Example: $(\neg a \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg c \vee d) \wedge (\neg b \vee \neg d)$
  $a = 1$ implies $c = 1$ (through c_1) and $d = 1$ (through c_3): same level
  Next independent choice would be one level down

Techniques in modern solvers

- *Decision level* for each variable assignment
  if assigned through BCP, same decision level
  Example: $(\neg a \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg c \vee d) \wedge (\neg b \vee \neg d)$
  $a = 1$ implies $c = 1$ (through c_1) and $d = 1$ (through c_3): same level
  Next independent choice would be one level down
- *Implication Graph*: tracks assignments implied by *BCP*

Techniques in modern solvers

- *Decision level* for each variable assignment
  if assigned through BCP, same decision level
  Example: $(\neg a \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg c \vee d) \wedge (\neg b \vee \neg d)$
  $a = 1$ implies $c = 1$ (through c_1) and $d = 1$ (through c_3): same level
  Next independent choice would be one level down
- *Implication Graph*: tracks assignments implied by *BCP*
- *Learned Clauses*: by analyzing implication graph

Watch Literals: Speeding up BCP

Assigning a literal $\Rightarrow$ scanning for affected clauses (expensive)

Most useful if propagation yields new assignment (unit clause)

Watch Literals: Speeding up BCP

Assigning a literal $\Rightarrow$ scanning for affected clauses (expensive)

Most useful if propagation yields new assignment (unit clause)

Idea: keep *two watch literals* per clause; link from literal to watched clause(s)

Watch Literals: Speeding up BCP

Assigning a literal $\Rightarrow$ scanning for affected clauses (expensive)

Most useful if propagation yields new assignment (unit clause)

Idea: keep *two watch literals* per clause; link from literal to watched clause(s)

For every clause where a watch literal becomes false:
  if nothing is left $\Rightarrow$ conflict
  if one literal left (unit clause) $\Rightarrow$ BCP
  else pick a new watched literal !

Special case: 2-SAT

Two literals per clause:
$(\neg a \vee \neg b) \wedge (b \vee c) \vee (\neg c \vee a) \wedge (\neg c \vee e) \wedge (f \vee g) \wedge (\neg g \vee h)$

Is this simpler?

Special case: 2-SAT

Two literals per clause:
$(\neg a \vee \neg b) \wedge (b \vee c) \vee (\neg c \vee a) \wedge (\neg c \vee e) \wedge (f \vee g) \wedge (\neg g \vee h)$

Is this simpler?

Yes, can do polynomial-time algorithm.

Repeat propagating one assignment until this stops.
Remaining part is independent $\Rightarrow$ may need to retry, but not backtrack

Or, graph with all literals x and $\neg x$ as nodes,
Clause $\ell_1 \vee \ell_2$ yields edges $\neg\ell_1 \to \ell_2$ and $\neg\ell_2 \to \ell_1$.
Check that no x and $\neg x$ in same strongly connected component.

Key Takeaways

SAT checking is a problem with many practical applications

and theoretical importance (NP-completeness)

Basic algorithm relies on just a few simple rules

- unit clause / one-literal rule
- boolean constraint propagation
- case splitting

Many optimizations in state-of-the-art solvers