H250: Honors Colloquium – Introduction to Computation

Satisfiability Checking

Marius Minea
marius@cs.umass.edu
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?

\[(\neg p \land \neg q) \land (\neg r \lor \neg (p \lor \neg (\neg r \rightarrow q)))\]

Maybe not at first sight, but can do it *systematically*
Option 1: try all cases

Formula: \((\neg p \land \neg q) \land (\neg r \lor \neg(p \lor \neg(\neg r \to q)))\)

\(p = T, q = T, r = T: \quad \text{is } false\)

\(p = T, q = T, r = F: \quad \text{is } false\)

...  
in effect, build \textit{truth table} until  
a \textit{true} value is found
or we exhaust all options
Option 2: case split and simplify

Formula: \( f = (\neg p \land \neg q) \land (\neg r \lor \neg (p \lor \neg (\neg r \rightarrow q))) \)

Let’s try \( p = T \).

\[
\begin{align*}
f \mid_{p=T} &= (F \land \neg q) \land (\neg r \lor \neg (T \lor \neg (\neg r \rightarrow q))) \\
&= F \land (\neg r \lor \neg (T \lor \neg (\neg r \rightarrow q))) \\
&= F
\end{align*}
\]
not successful, but saves us from trying all subcases for \( q \) and \( r \)
Option 2: case split and simplify

Formula: \( f = (\neg p \land \neg q) \land (\neg r \lor \neg (p \lor \neg (\neg r \rightarrow q))) \)

Let's try \( p = T \).

\[
f \mid_{p=T} = (F \land \neg q) \land (\neg r \lor \neg (T \lor \neg (\neg r \rightarrow q)))
\]

\[
= F \land (\neg r \lor \neg (T \lor \neg (\neg r \rightarrow q)))
\]

\[
= F
\]

not successful, but saves us from trying all subcases for \( q \) and \( r \).

We try \( p = F \).

\[
f \mid_{p=F} = (T \land \neg q) \land (\neg r \lor \neg (F \lor \neg (\neg r \rightarrow q)))
\]

\[
= \neg q \land (\neg r \lor \neg (\neg r \rightarrow q))
\]

\[
= \neg q \land (\neg r \lor r \lor q)
\]

\[
= \neg q
\]

and the formula is true for \( p = q = F \), 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 \( \neg (f_{\text{orig}} \leftrightarrow f_{\text{opt}}) \) 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 familiar puzzles: n-queens, Sudoku, etc.
A motivating puzzle

Can you find a sequence of 8 bits $b_1, b_2, \ldots, b_8$ 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

\[(b_1 \lor b_2 \lor b_3) \land (\neg b_1 \lor \neg b_2 \lor \neg b_3)\]

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)

\[\ldots \land (b_1 \lor b_4 \lor b_7) \land (\neg b_1 \lor \neg b_4 \lor \neg b_7)\]
\[\land (b_2 \lor b_5 \lor b_8) \land (\neg b_2 \lor \neg b_5 \lor \neg b_8)\]

We have a formula structured as a conjunction of constraints
common in constraint satisfaction problems
A formula is in conjunctive normal form, if it is a conjunction $\land$ of clauses, where a clause is a disjunction $\lor$ 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 ORs

$$\begin{align*}
(a \lor \neg b \lor \neg d) \\
\land (\neg a \lor \neg b) \\
\land (\neg a \lor c \lor \neg d) \\
\land (\neg a \lor b \lor c)
\end{align*}$$

Can you see a way to make the formula true?
Advantages of CNF

\( (a \lor \neg b \lor \neg d) \)
\( \land (\neg a \lor \neg b) \)
\( \land (\neg a \lor c \lor \neg d) \)
\( \land (\neg a \lor b \lor c) \)

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)

\[
\begin{align*}
a \\
in \land (\neg a \lor b \lor c) \\
\land (\neg a \lor \neg b \lor \neg c)
\end{align*}
\]

\[
\begin{align*}
(a \lor b) \\
in \land \neg b \\
\land (\neg a \lor \neg b \lor c)
\end{align*}
\]

$a$ must be taken $T$

$b$ must be taken $F$

otherwise the formula is false
Rules for satisfiability: Unit clause

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

\[
\begin{align*}
& a \\
& \text{in } \land (\neg a \lor b \lor c) \\
& \land (\neg a \lor \neg b \lor \neg c)
\end{align*}
\]

\((a \lor b)\)

\[
\begin{align*}
& \text{in } \land \neg b \\
& \land (\neg a \lor \neg b \lor c)
\end{align*}
\]

\( a \) must be taken \( T \)

\( b \) must be taken \( F \)

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

\( \text{R}_{\text{BCP1}}: \)  If a literal \( L \) is \( T \), \textit{delete clauses containing} \( L \) they are true (as desired), no need to consider further

\( \text{R}_{\text{BCP2}}: \)  If a literal \( L \) is \( F \), \textit{delete it} from all clauses \textit{false} can’t help make clause \( true \)

Previous examples simplify:

\[
\begin{align*}
  & a \\
  & \land \ (\neg a \lor b \lor c) \\
  & \land \ (\neg a \lor \neg b \lor \neg c)
\end{align*}
\]

\[
\begin{align*}
  & \rightarrow \ a = T \\
  & (b \lor c) \\
  & \land \ (\neg b \lor \neg c)
\end{align*}
\]

\[
\begin{align*}
  & \ (a \lor b) \\
  & \land \ \neg b \\
  & \land \ (\neg a \lor \neg b \lor c) \checkmark
\end{align*}
\]

and from here \( a = T \), formula is satisfiable
Rules for satisfiability: Stopping

\[ R_{STOP}: \] If there are \textit{no more clauses}, formula is satisfiable with the truth assignment obtained so far.

If we have an \textit{empty clause}, formula is not satisfiable, no literals in empty clause to make it true.

\[(a \lor b)\checkmark \land a\checkmark \land (a \lor \neg b \lor c)\checkmark\]

\[ b\checkmark \land (\neg b \lor c) \land (\neg b \lor \neg c) \rightarrow c = T \rightarrow \emptyset \text{ UNSAT}\]

\[ c = T \text{ makes } \neg c \text{ empty clause } \Rightarrow \text{ not satisfiable} \]
Rules for satisfiability: Stopping

**R\textsubscript{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

\[(a \lor b) \checkmark\]
\[\land a \checkmark \quad a=\top \quad SAT\]
\[\land (a \lor \neg b \lor c) \checkmark\]

\[b \checkmark\]
\[\land (\neg b \lor c) \quad b=\top \quad \land\neg b \lor c \checkmark \quad c=\top \quad \emptyset \quad UNSAT\]
\[\land (\neg b \lor \neg c)\]

\(c = \top\) 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 \(\checkmark\)
- empty clause means *UNSAT*: no literals to make it true
Rules for satisfiability: Case splitting

What if no more simplifications can be done?

\[ a \land (\neg a \lor b \lor c) \land (\neg b \lor \neg c) \quad \overset{a=T}{\longrightarrow} \quad (b \lor c) \land (\neg b \lor \neg c) \quad ?? \]

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

Rule $R_{\text{STOP}}$ tells us we are done (have an answer)

Rule $R_{\text{CASE}}$ 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 $R_{\text{STOP}}$)
Writing a pseudocode algorithm

We’ll either return the set of literals assigned \( T \) (can use to check) or raise an exception \( \text{Unsat} \)

```
function solve(clauses: clause list, truelit: lit set)
  while clauses contains a unit clause do
    (clauses, truelit) = simplify(clauses, truelit) (* RUNIT, RBCP*)
```
Writing a pseudocode algorithm

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

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

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

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

```pseudocode
function solve(clauses: clause list, truelit: lit set)
while clauses contains a unit clause do
    (clauses, truelit) = simplify(clauses, truelit) (* RUNIT, RBCP*)
if clauses = empty list then
    return truelit (* RSTOP: SAT, return set of true literals *)
if clauses contains empty clause then
    raise Unsat (* RSTOP: UNSAT *)
```
Writing a pseudocode algorithm

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

```plaintext
function solve(clauses: clause list, truelit: lit set)
  while clauses contains a unit clause do
    (clauses, truelit) = simplify(clauses, truelit) (* RUNIT, RBCP*)
  if clauses = empty list then
    return truelit (* RSTOP: SAT, return set of true literals *)
  if clauses contains empty clause then
    raise Unsat (* RSTOP: UNSAT *)
  choose a literal p
  try solve (clauses, truelit ∪ {¬p}) (* RCASE: try p=F *)
  with Unsat → solve (clauses, truelit ∪ {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: \[\emptyset\]

\[-x_1 \lor x_2\]
\[x_3 \lor x_4 \lor x_5\]
\[x_1\]
\[x_1 \lor x_7\]
\[-x_1 \lor -x_4\]
\[-x_3 \lor x_5\]
\[-x_5 \lor x_6\]

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: \[ \emptyset \]

\[ \Rightarrow \neg x_1 \lor x_2 \]
\[ x_3 \lor x_4 \lor x_5 \]
\[ x_1 \]
\[ x_1 \lor x_7 \]
\[ \neg x_1 \lor \neg x_4 \]
\[ \neg x_3 \lor x_5 \]
\[ \neg x_5 \lor x_6 \]

No true literals yet, can’t simplify, just copy
Applying Boolean Constraint Propagation

True literals:

\[\neg x_1 \lor x_2\]
\[x_3 \lor x_4 \lor x_5\]
\[x_1\]
\[x_1 \lor x_7\]
\[\neg x_1 \lor \neg x_4\]
\[\neg x_3 \lor x_5\]
\[\neg x_5 \lor x_6\]

\[\emptyset\]

\[\neg x_1 \lor x_2\]
\[x_3 \lor x_4 \lor x_5\]

No true literals yet, can’t simplify, just copy
Applying Boolean Constraint Propagation

**True literals:**

\[ \emptyset \]

- \( \neg x_1 \lor x_2 \)
- \( x_3 \lor x_4 \lor x_5 \)

\( \Rightarrow x_1 \)

- \( x_1 \lor x_7 \)
- \( \neg x_1 \lor \neg x_4 \)
- \( \neg x_3 \lor x_5 \)
- \( \neg x_5 \lor x_6 \)

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:  \[
\emptyset \quad \{1\}
\]

\[\neg x_1 \lor x_2 \quad \neg x_1 \lor x_2\]
\[x_3 \lor x_4 \lor x_5 \quad x_3 \lor x_4 \lor x_5\]
\[x_1 \quad x_1 \lor x_7\]
\[\neg x_1 \lor \neg x_4 \quad \neg x_3 \lor x_5 \quad \neg x_5 \lor x_6\]

Simplify clauses accumulated so far with \(x_1\) set to T
Applying BCP: Eagerly re-simplify with unit clause

True literals:

\[ \emptyset \quad \{1\} \]

\[ \neg x_1 \lor x_2 \quad \Rightarrow \quad \neg x_1 \lor x_2 \quad x_2 \]

\[ x_3 \lor x_4 \lor x_5 \quad x_3 \lor x_4 \lor x_5 \]

\[ x_1 \]

\[ x_1 \lor x_7 \]

\[ \neg x_1 \lor \neg x_4 \]

\[ \neg x_3 \lor x_5 \]

\[ \neg x_5 \lor x_6 \]

\[ \neg x_1 \text{ is filtered from first clause} \]

New unit clause \( x_2 \) is found
Applying BCP: continue with two true literals

True literals: \( \emptyset \) \( \{1, 2\} \)

\(-x_1 \lor x_2 \quad \Rightarrow \quad x_3 \lor x_4 \lor x_5 \quad x_3 \lor x_4 \lor x_5 \)

\(x_3 \lor x_4 \lor x_5\)

\(x_1\)

\(x_1 \lor x_7\)

\(-x_1 \lor -x_4\)

\(-x_3 \lor x_5\)

\(-x_5 \lor x_6\)

Done at this level, next clause not changed
Applying BCP: return to first-level traversal

True literals: \{1, 2\}

\neg x_1 \lor x_2 \quad x_3 \lor x_4 \lor x_5
x_3 \lor x_4 \lor x_5
x_1
⇒ x_1 \lor x_7 \checkmark
\neg x_1 \lor \neg x_4
\neg x_3 \lor x_5
\neg x_5 \lor x_6

Clause is true, ignored
Applying BCP: return to first-level traversal

True literals: \{1, 2\}

\neg x_1 \lor x_2 \\
x_3 \lor x_4 \lor x_5 \\
x_1 \\
x_1 \lor x_7 \checkmark \\
\implies \neg x_1 \lor \neg x_4 \\
\neg x_3 \lor x_5 \\
\neg x_5 \lor x_6 \\

New unit clause \neg x_4 found
Applying BCP: restart simplification

True literals:

\( \neg x_1 \lor x_2 \)
\( x_3 \lor x_4 \lor x_5 \)
\( x_1 \)
\( x_1 \lor x_7 \)
\( \neg x_1 \lor \neg x_4 \)

\( \neg x_3 \lor x_5 \)
\( \neg x_5 \lor x_6 \)

\( \Rightarrow x_3 \lor x_4 \lor x_5 \)

Only clause in list simplifies to \( x_3 \lor x_5 \)
Applying BCP: continue at first level

**True literals:** \( \{1, 2, -4\} \)

\(-x_1 \lor x_2\)  \(\Rightarrow\)  \(-x_3 \lor x_5\)

\(x_3 \lor x_4 \lor x_5\)  \(\Rightarrow\)  \(-x_3 \lor x_5\)

\(x_1\)

\(x_1 \lor x_7\)

\(-x_1 \lor -x_4\)

\(-x_3 \lor x_5\)

\(-x_5 \lor x_6\)
Applying BCP: continue at first level

True literals: \{1, 2, −4\}

\[
\begin{align*}
\neg x_1 & \lor x_2 \\
x_3 & \lor x_4 \lor x_5 \\
x_1 & \\
x_1 & \lor x_7 \\
\neg x_1 & \lor \neg x_4 \\
\neg x_3 & \lor x_5 \\
\Rightarrow & \quad \neg x_5 \lor x_6 \\
\end{align*}
\]

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 \textit{sequence of 8 bits} $b_1, b_2, \ldots, b_8$ that has

- no three equally-spaced 0s
- no three equally-spaced 1s

All clauses are triples (minus denotes negated literal)

\[
\begin{align*}
[1; 2; 3]; & [-1; -2; -3]; [2; 3; 4]; [-2; -3; -4]; \\
[3; 4; 5]; & [-3; -4; -5]; [4; 5; 6]; [-4; -5; -6]; \\
[5; 6; 7]; & [-5; -6; -7]; [6; 7; 8]; [-6; -7; -8]; \\
[1; 3; 5]; & [-1; -3; -5]; [2; 4; 6]; [-2; -4; -6]; \\
[3; 5; 7]; & [-3; -5; -7]; [4; 6; 8]; [-4; -6; -8]; \\
[1; 4; 7]; & [-1; -4; -7]; [2; 5; 8]; [-2; -5; -8] \\
\end{align*}
\]

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
⇒ Heuristics for better choice
Limitations of basic DPLL

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

- **Does not learn** from conflicts: a bad partial assignment may be retried multiple times (in different subtrees)

- **Clause learning**: learn new clauses (constraints) that need to be satisfied, e.g., \((A \lor p) \land (B \lor \neg p) \rightarrow (A \lor B)\)

- Backtracks one level, but root of conflict may be higher up
  ⇒ non-chronological backtracking: several levels up, avoids pointless search

- **Conflict Driven Clause Learning**: technique used in family of modern solvers
Limitations of basic DPLL

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

- **Does not learn** from conflicts: a bad partial assignment may be retried multiple times (in different subtrees)
  ⇒ *clause learning*: learn new clauses (constraints) that need to be satisfied

quad e.g., \((A \lor p) \land (B \lor \neg p) \rightarrow (A \lor B)\)
Limitations of basic DPLL

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

- **Does not learn** from conflicts: a bad partial assignment may be retried multiple times (in different subtrees) ⇒ *clause learning*: learn new clauses (constraints) that need to be satisfied
  quad e.g., \((A \lor p) \land (B \lor \neg p) \rightarrow (A \lor B)\)

- **Backtracks one level**, but root of conflict may be higher up
Limitations of basic DPLL

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

- **Does not learn** from conflicts: a bad partial assignment may be retried multiple times (in different subtrees)
  ⇒ *clause learning*: learn new clauses (constraints) that need to be satisfied
  quad e.g., \((A \lor p) \land (B \lor \neg p) \Rightarrow (A \lor B)\)

- **Backtracks one level**, but root of conflict may be higher up
  ⇒ *non-chronological backtracking*: several levels up, avoids pointless search
Limitations of basic DPLL

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

```plaintext
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: \((\neg a \lor c) \land (\neg a \lor b \lor \neg c) \land (\neg c \lor d) \land (\neg b \lor \neg d)\)
  
  \(a = 1\) 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: \((\neg a \lor c) \land (\neg a \lor b \lor \neg c) \land (\neg c \lor d) \land (\neg b \lor \neg d)\)
  
  \(a = 1\) 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: \((\neg a \lor c) \land (\neg a \lor b \lor \neg c) \land (\neg c \lor d) \land (\neg b \lor \neg d)\)
  \(a = 1\) 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
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:
\[(\neg a \lor \neg b) \land (b \lor c) \lor (\neg c \lor a) \land (\neg c \lor e) \land (f \lor g) \land (\neg g \lor h)\]

Is this simpler?
Special case: 2-SAT

Two literals per clause:
\[(\neg a \lor \neg b) \land (b \lor c) \lor (\neg c \lor a) \land (\neg c \lor e) \land (f \lor g) \land (\neg g \lor h)\]

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 \(\ell_1 \lor \ell_2\) yields edges \(\neg \ell_1 \rightarrow \ell_2\) and \(\neg \ell_2 \rightarrow \ell_1\).
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