CSE 417 Autumn 2025

Lecture 26: How SAT solvers work

Glenn Sun

Review of last lecture

SAT (Satisfiability)

literals: variables or their negation

$$\boldsymbol{a}, \neg \boldsymbol{b}, \boldsymbol{x}, \neg \boldsymbol{x}, \boldsymbol{y}$$

clause: OR of literals

$$(\boldsymbol{a} \lor \neg \boldsymbol{b}), (\boldsymbol{x} \lor \neg \boldsymbol{y} \lor \boldsymbol{z})$$

conjunction normal form (CNF): AND of clauses

$$(\boldsymbol{a} \lor \neg \boldsymbol{b}) \land (\boldsymbol{x} \lor \neg \boldsymbol{y} \lor \boldsymbol{z})$$

SAT (Satisfiability)

Input: A CNF formula $f(x_1, \dots, x_n)$ (equivalently a set of clauses)

Goal: Does there exist $x_1, \dots, x_n$ such that $f(x_1, \dots, x_n)$ is true?

SAT is NP-hard: We don't believe we can solve it quickly in general.

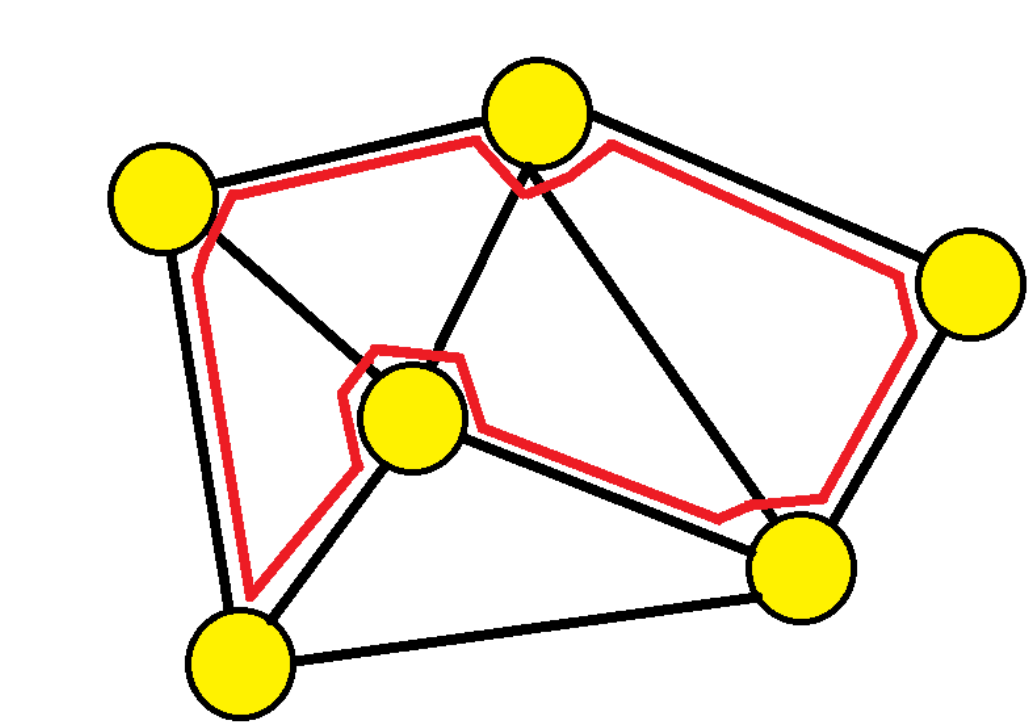
However, in the last ~20 years, we've gotten very good!

Real world problems with >1,000,000 variables/clauses are OK!

Hamiltonian path

Input: An undirected graph with vertices $\boldsymbol{V}$ and edges $\boldsymbol{E}$

Goal: Is there a path that uses every vertex exactly once?



Also an NP-hard problem!

Hamiltonian path

Define $p_{v,i}$ to mean "vertex v is the ith vertex on the path".

- Each vertex appears exactly once on the path.
- Each position on the path has exactly one vertex.
- If two vertices are adjacent on the path, then there is an edge between them.

"Exactly one" constraints

We just saw this with "every vertex gets exactly one color"!

To translate "exactly one of $x_1, x_2, \dots, x_n$":

- At least one of $x_1, x_2, \dots, x_n$:

  $$(x_1 \lor x_2 \lor \cdots \lor x_n)$$

- No two of of $x_1, x_2, \dots, x_n$:

  $(\neg x_i \lor \neg x_j)$ for every pair of i and j

Requires $O(n^2)$ constraints.

Hamiltonian path

Define $p_{v,i}$ to mean "vertex v is the ith vertex on the path".

- Each vertex appears exactly once on the path.

  "exactly one of $p_{v,1}, \dots, p_{v,n}$" for every vertex v

- Each position on the path has exactly one vertex.

  "exactly one of $p_{1,i}, \dots, p_{n,i}$" for every position i

Hamiltonian path

Define $p_{v,\,i}$ to mean "vertex v is the ith vertex on the path".

- If two vertices are adjacent on the path, then there is an edge between them.

$$(p_{u,\,i} \wedge p_{v,\,i+1}) \Rightarrow \text{“}(u, v) \text{ is an edge”}$$

Use contrapositive: whenever (u, v) is not an edge, include clause

$$(\neg p_{u,\,i} \vee \neg p_{v,\,i+1})$$

Tseitin transformations

To translate longer Boolean sentences efficiently, introduce helper variables! For example, if you have $x_1 \oplus x_2 \oplus \cdots \oplus x_n$, let

- $z_2 \Leftrightarrow x_1 \oplus x_2$
- $z_3 \Leftrightarrow z_2 \oplus x_3$
- …
- $z_n \Leftrightarrow z_{n-1} \oplus x_n$

Each $a \Leftrightarrow b \oplus c$ takes 4 clauses to convert to CNF (from concept check).

Use $n - 1$ new variables and represent this sentence in $O(n)$ clauses!

Tseitin transformations

For ***any*** boolean operation $\boldsymbol{R}$ (could be XOR, AND, OR, etc.),

$$\boldsymbol{a} \iff \boldsymbol{b}\ \boldsymbol{R}\ \boldsymbol{c}$$

takes at most 8 clauses to convert to CNF (since there are only 8 possible clauses at all with 3 variables).

Doing this is called a **Tseitin transformation**.

Program verification

Input: A computer program written in some language and a formal specification

Goal: Does the program meet the spec for all inputs?

Program verification

```
division(int x, int y) {
    int r = x;
    int q = 0;

    while (r >= y) {
        r = r - y;
        q++;
    }

    assert x == y * q + r;
    assert r >= 0 && r < Math.abs(y);
```

code

spec

Program verification

The basic idea is to convert our program into **single static assignment form**, where it will only have:

- Basic functions like +, >, and _ ? _ : _
- Assignment to variables
- "Assume" statements
- "Assert" statements

Kind of like OR, AND, XOR, etc?

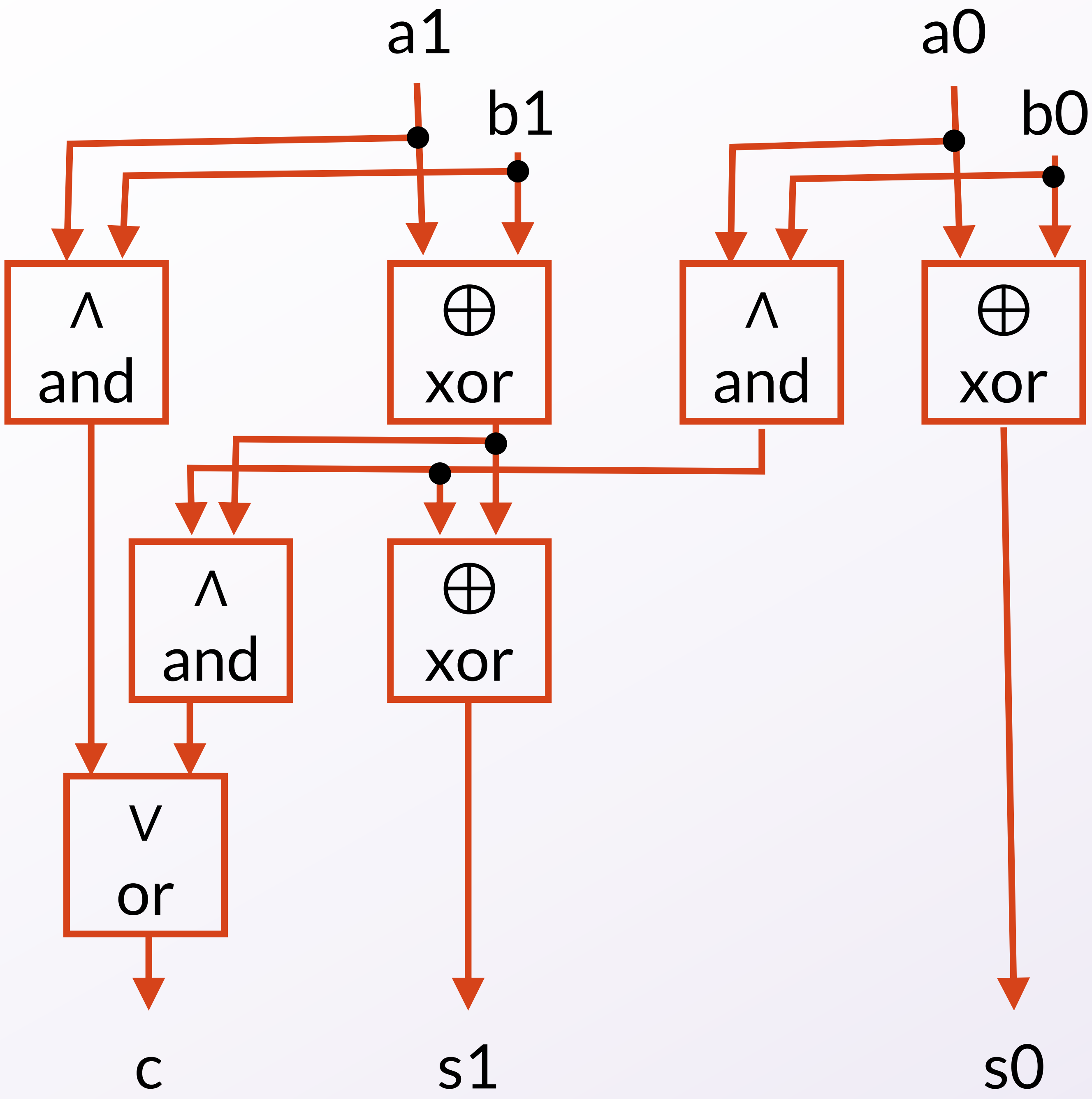
Kind of like Tseitin transformations, except not just bools!

"The spec is satisfied for all small inputs that don't result in long loops."

Circuits

In a class on digital design (CSE 369/EE 271), you would learn how to implement all these basic functions with circuits!

	a1	a0
+	b1	b0
c	s1	s0



Some things you can do with circuits

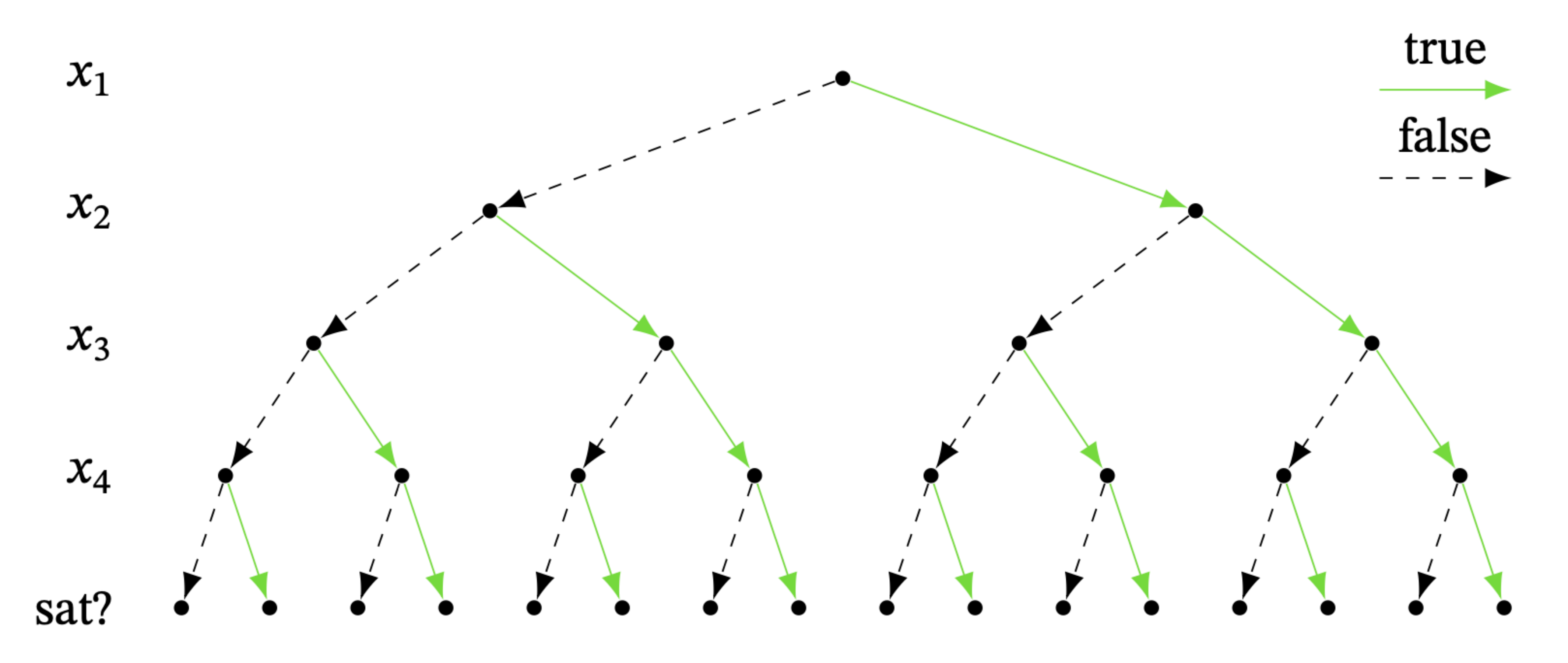
Addition of n-bit binary numbers	$O(n)$ gates
Multiplication of n-bit binary numbers	$O(n^2)$ gates with simple implementation, improvable
Comparison of n-bit numbers	$O(n)$ gates
If-then-else for n-bit numbers	$O(n)$ gates
Anything that you can compute on a computer in T time	at most $O(T \log T)$ gates

By Tseitin transformations: # new clauses ≈ # new variables = # gates!

The DPLL algorithm

First idea: brute force

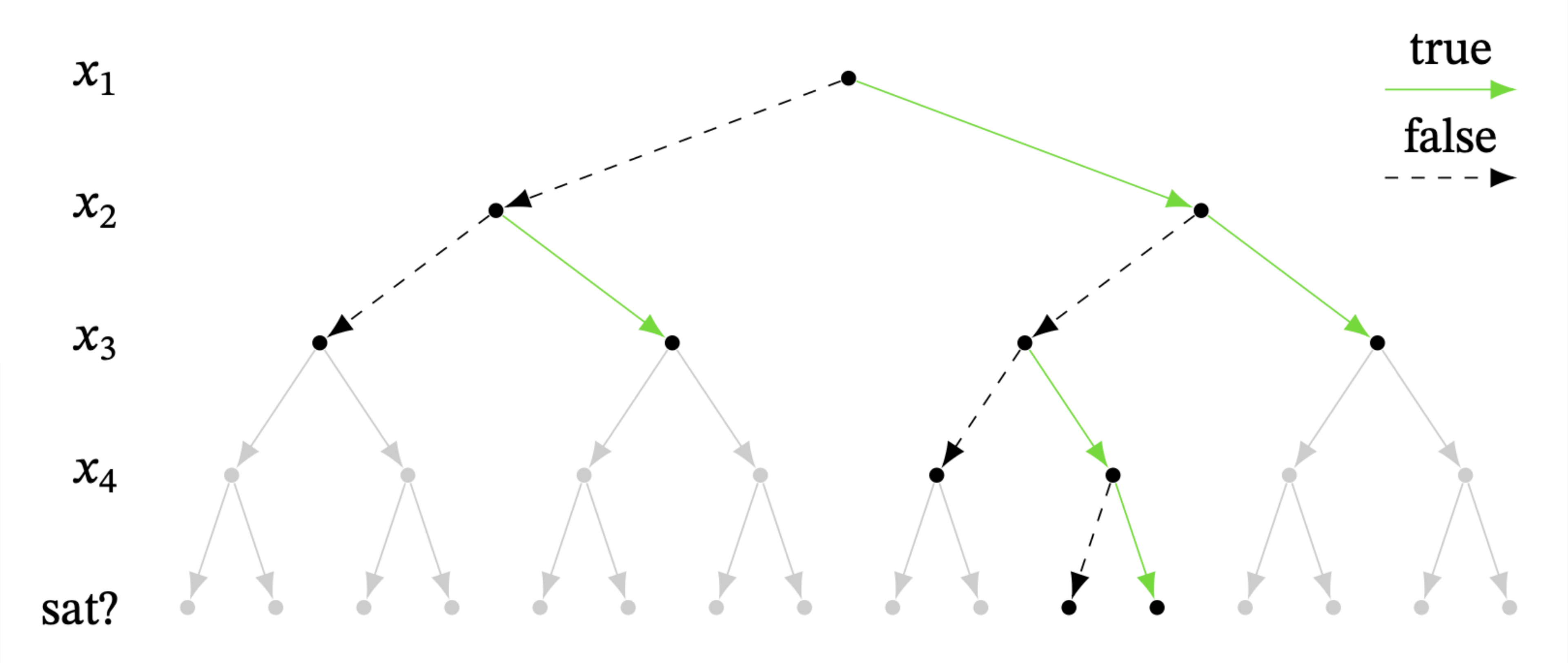
To solve SAT via brute force, we try every assignment.



Second idea: check unsatisfiability earlier

At every node, check if we've already made the things unsatisfiable.

$$(x_1 \lor x_2) \quad (\neg x_2) \quad (x_2 \lor x_3) \quad (x_2 \lor \neg x_3 \lor x_4) \quad (\neg x_3 \lor \neg x_4).$$



Even faster: Your ideas

$$(x_1 \vee x_2) \qquad (\neg x_2) \qquad (x_2 \vee x_3) \qquad (x_2 \vee \neg x_3 \vee x_4) \qquad (\neg x_3 \vee \neg x_4).$$

"As a human, I would find the clauses with only one variable, which could easily tell whether the variable should be true or false."

"I would say at least one of x_1 and x_2 needs to be true. Then the second says that x_2 has to be false, so then plugging that back in, x_1 needs to be true."

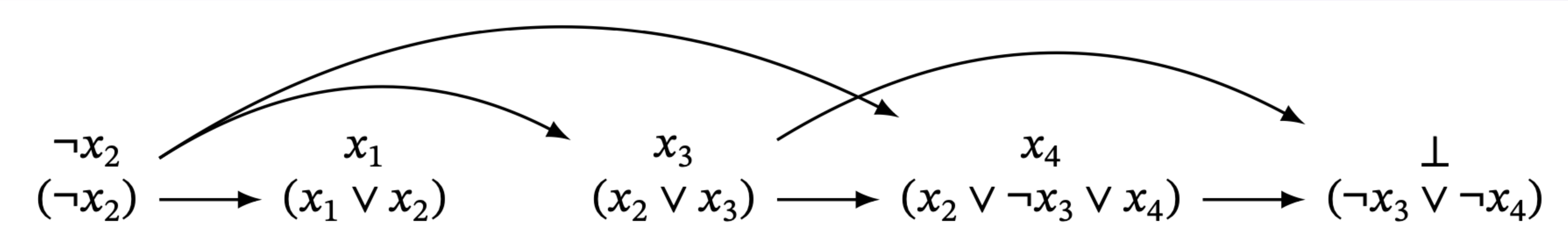
"If x_2 is false then $(x_1 \vee x_2)$ forces x_1 true and $(x_2 \vee x_3)$ forces x_3 true."

Unit propagation

A **unit** is a clause with one literal.

If you know a unit, simplify other clauses with this knowledge!

$$(x_1 \vee x_2) \qquad (\neg x_2) \qquad (x_2 \vee x_3) \qquad (x_2 \vee \neg x_3 \vee x_4) \qquad (\neg x_3 \vee \neg x_4).$$



Unit propagation

Input: set of units $\boldsymbol{U}$ and a clause $\boldsymbol{C}$

1. if every literal in $\boldsymbol{C}$ is made false by $\boldsymbol{U}$,
2.   return unsatisfiable
3. else if every literal in $\boldsymbol{C}$ except one is made false by $\boldsymbol{U}$,
4.   return the unfalsified literal in $\boldsymbol{C}$

Unit propagation

Input: set of units $\boldsymbol{U}$ and set of clauses Δ

1. Repeat the following until an entire iteration passes without propagating a new unit:
2.   **for each** clause $\boldsymbol{C} \in \Delta$,
3.   Unit propagate with $\boldsymbol{U}$ and $\boldsymbol{C}$, possibly updating $\boldsymbol{U}$.
4.   **if** unit propagation returned “unsatisfiable”,
5.   **return** “unsatisfiable”
6. **return** the updated set of units $\boldsymbol{U}$

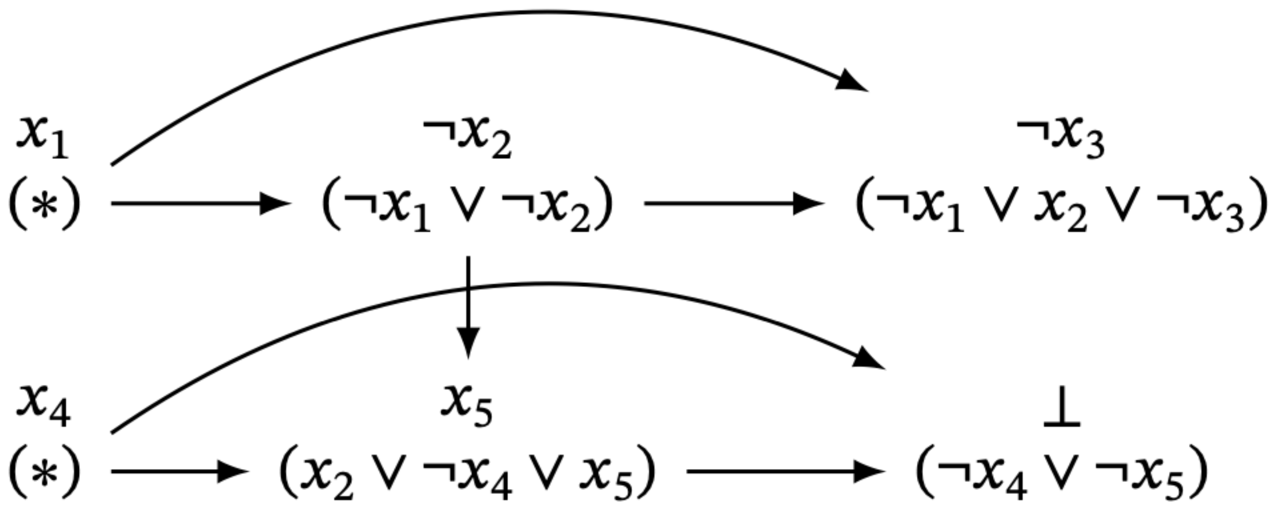
DPLL algorithm (Davis–Putnam–Logemann–Loveland, 1961)

The following defines a recursive function $\mathbf{DPLL}(\boldsymbol{U}, \boldsymbol{\Delta})$

1. Run unit propagation and update $\boldsymbol{U}$.
2. **if** unit propagation learns contradiction, **return** false
3. **else**,
4.   **if** $\boldsymbol{U}$ does not set every variable,
5.   Pick an unset variable $\boldsymbol{x}$.
6.   **return** $\mathbf{DPLL}(\boldsymbol{U} \cup \{\boldsymbol{x}\}, \boldsymbol{\Delta})$ OR $\mathbf{DPLL}(\boldsymbol{U} \cup \{\neg \boldsymbol{x}\}, \boldsymbol{\Delta})$
7.   **else**, **return** true

DPLL algorithm

$$(\neg x_1 \lor \neg x_2) \quad (\neg x_1 \lor x_2 \lor \neg x_3) \quad (x_2 \lor \neg x_4 \lor x_5) \quad (\neg x_4 \lor \neg x_5)$$

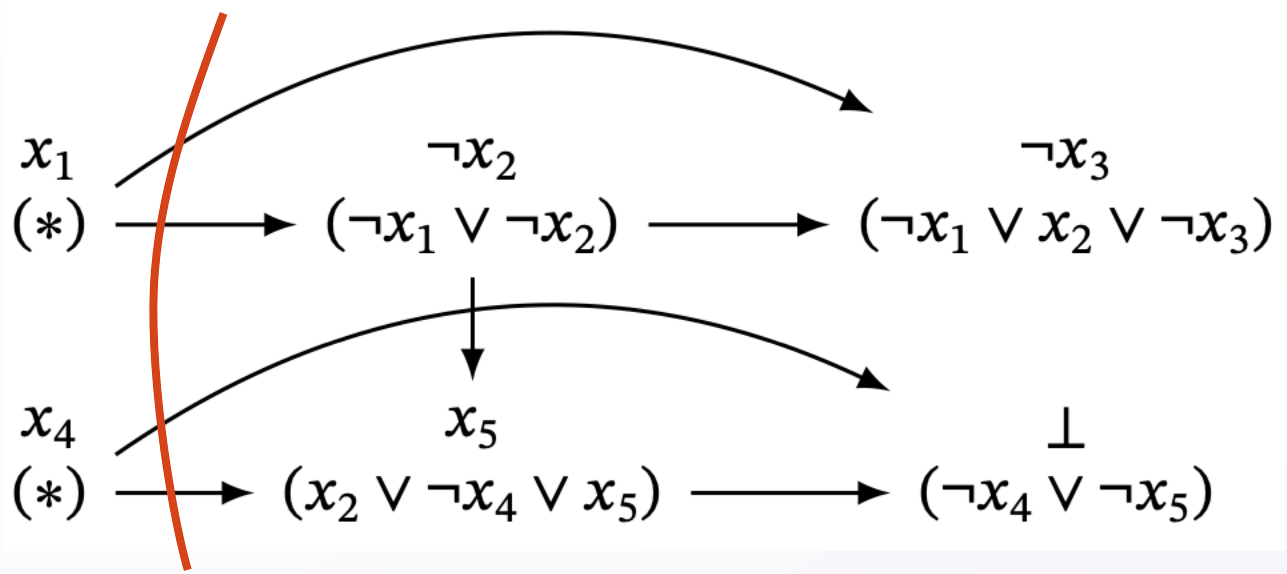


Next, try x_4 being false.

In this case, these clauses are now all satisfied!

The CDCL algorithm

Cuts in an implication graph



This cut says: if we know x_1 and x_4, then we get a contradiction.

Therefore, $(\neg x_1 \vee \neg x_4)$ must be true.

Learning this clause lets unit propagation deduce $\neg x_4$ from x_1!

Clause learning view of DPLL

DPLL recursively calls:

$$\textbf{return } \mathbf{DPLL}(U \cup \{x\}, \Delta) \text{ OR } \mathbf{DPLL}(U \cup \{\neg x\}, \Delta)$$

Instead, it is equivalent for the solver to:

call this the DPLL clause

- Decide to check only $\mathbf{DPLL}(U \cup \{x\}, \Delta)$.
- Upon reaching contradiction,
  - Let C be the clause with $\neg x$ for every decision x.
  - Add C to Δ and revert U to before the last decision.

Clause learning view of DPLL

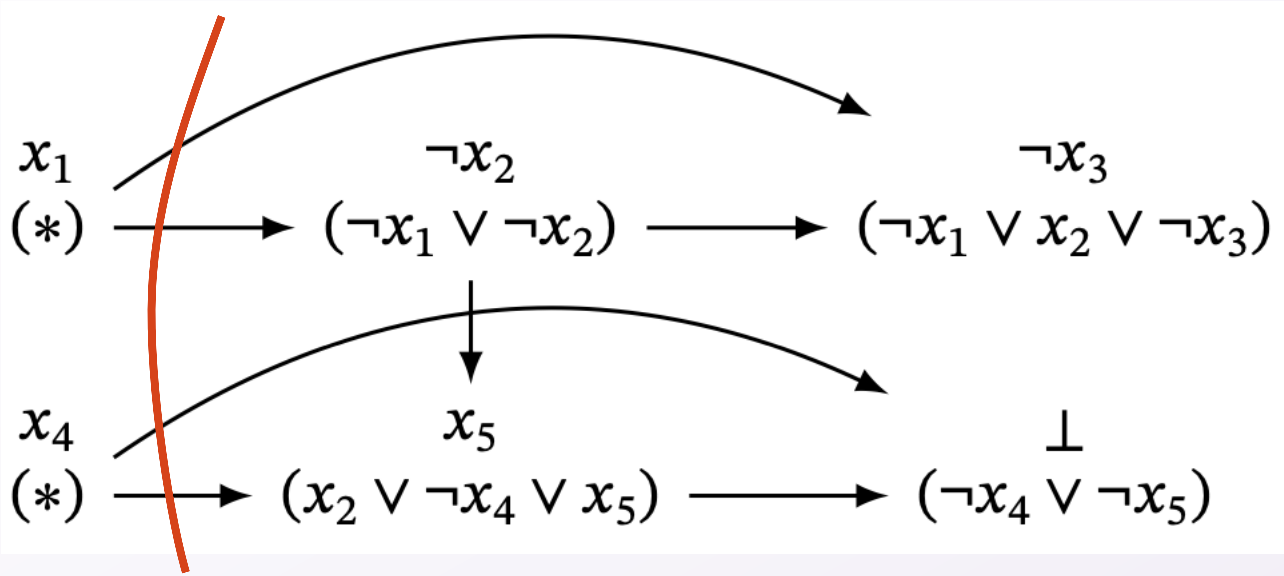
1. Do the following in a loop:
2. Run unit propagation and update U.
3. if unit propagation learns a contradiction,
4. if U still contains a decision, learn the DPLL clause and revert U to before the last decision.
5. else, return unsatisfiable
6. else,
7. if there is an unset variable, pick one and add it to U.
8. else, return satisfiable.

Cuts in an implication graph

Take any cut separating the decisions from the contradiction.

The **conflict clause** associated with this cut has the negation of everything immediately before the cut.

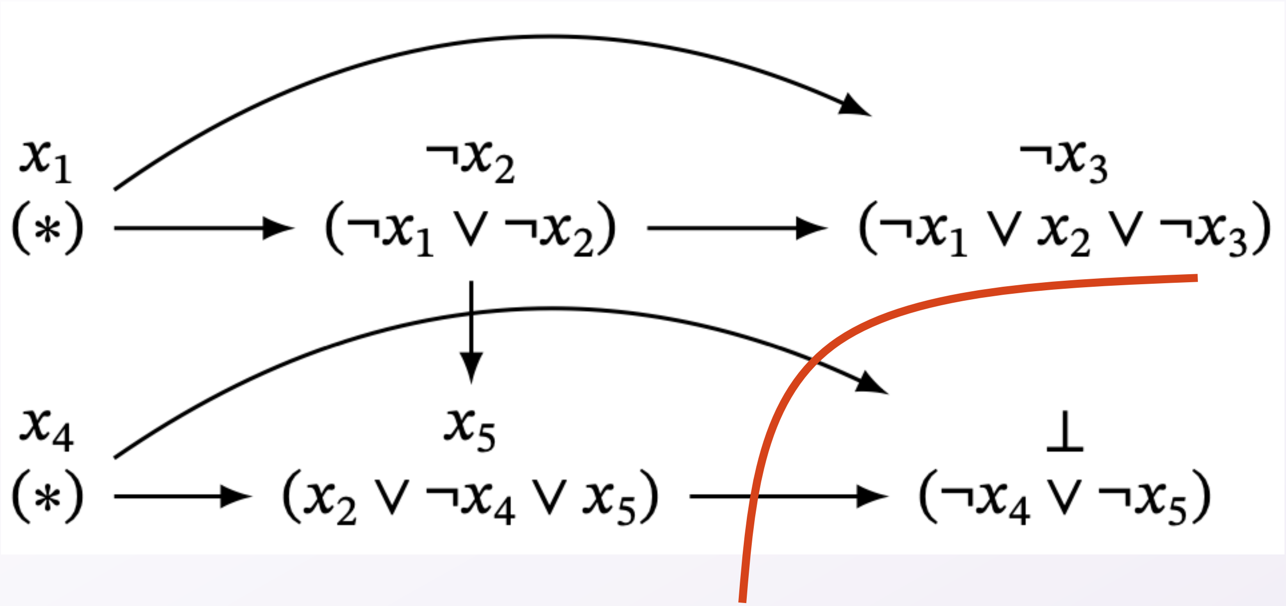
$(\neg x_1 \vee \neg x_4)$



Cuts in an implication graph

Take any cut separating the decisions from the contradiction.

The **conflict clause** associated with this cut has the negation of everything immediately before the cut.



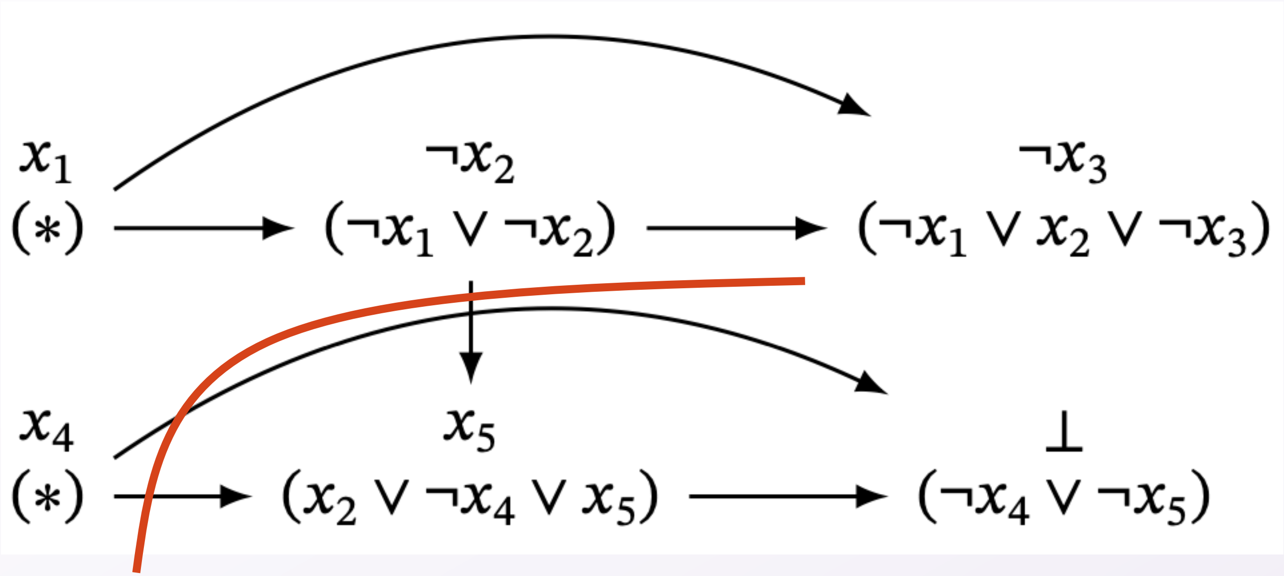
$(\neg \boldsymbol{x_4} \vee \neg \boldsymbol{x_5})$

Cuts in an implication graph

Take any cut separating the decisions from the contradiction.

The **conflict clause** associated with this cut has the negation of everything immediately before the cut.

$(\neg x_2 \vee \neg x_4)$

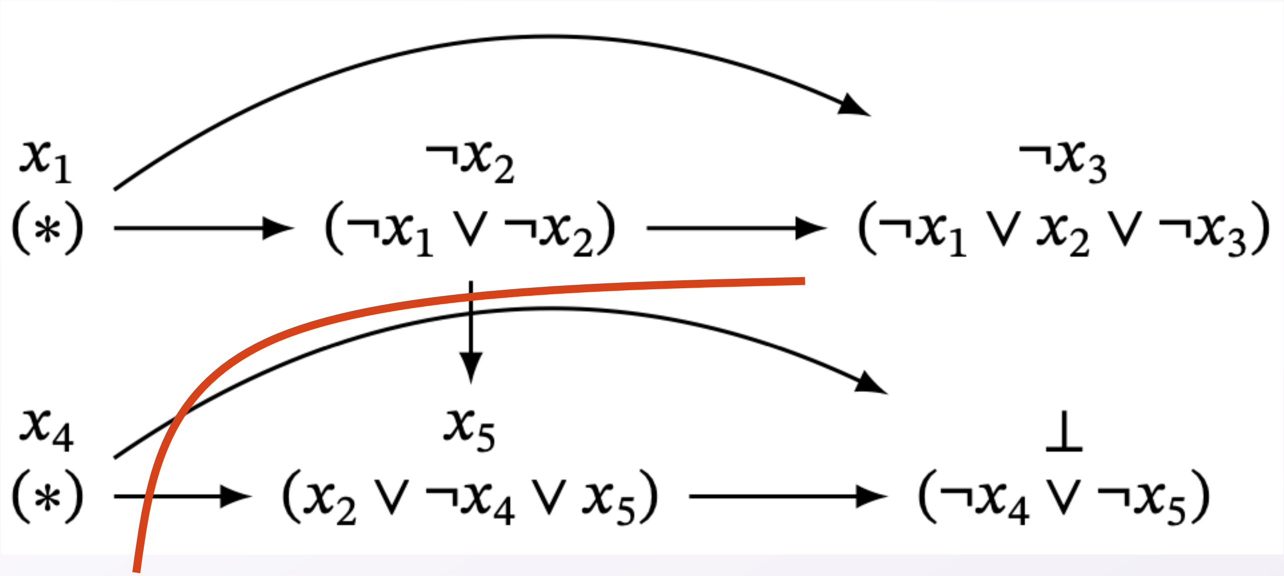


Cuts in an implication graph

A conflict clause is **asserting** if it can be used for unit propagation with fewer decisions.

Any asserting clause can be learned!

$(\neg x_2 \vee \neg x_4)$



CDCL (conflict-driven clause learning)

1. Do the following in a loop:
2. Run unit propagation and update U.
3. if unit propagation learns a contradiction,
4. if U still contains a decision, learn any asserting clause and revert U to before the last decision.
5. else, return unsatisfiable
6. else,
7. if there is an unset variable, pick one and add it to U.
8. else, return satisfiable.

More improvements

- Improved heuristics for choosing the next variable to branch on
- Improved heuristics for choosing which conflict clause to learn
- Faster unit propagation with watched literals
- Random restarts
- Clause deletion

Improving SAT solvers remains an active area of research.

Final reminders

HW6 (Greedy) resubmissions close tonight @ 11:59pm!

HW7 (Flows) due tonight @ 11:59pm!

I have OH now-12:30pm:

- Meet at front of classroom, we'll walk over together
- CSE (Allen) 214 if you're coming later

Nathan has online OH 12–1pm:

- https://washington.zoom.us/my/nathanbrunelle