Boolean Satisfiability

Discrete Math, Fall 2025

Konstantin Chukharev

Boolean Satisfiability

"Can machines think?"

— Alan Turing


Stephen Cook


Leonid Levin


Richard Karp


Marijn Heule


Armin Biere


João Marques-Silva

A Puzzle to Start

You're given a formula. Can you make it true?

$$(x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$$

Try $x = 1, y = 1, z = 1$:

- $(1 \vee 1)$ ✓
- $(0 \vee 1)$ ✓
- $(0 \vee 0)$ ✗

Try $x = 1, y = 0, z = 1$:

- $(1 \vee 0)$ ✓
- $(0 \vee 1)$ ✓
- $(1 \vee 0)$ ✓ **Found it!**

This is the **Boolean Satisfiability Problem** (SAT) — *the* central problem in computer science.

Why Should You Care?

$475 million bug

In 1994, Intel's Pentium had a floating-point division bug.

Now: Every Intel CPU is verified using SAT solvers.

1/3 of security bugs

Microsoft's SAGE tool found 1/3 of all Windows 7 security vulnerabilities.

How? By encoding programs as SAT formulas.

The paradox: SAT is proven to be "computationally hard" (NP-complete), yet modern solvers routinely handle formulas with *millions* of variables!

The SAT Problem

Definition 1: A formula φ is *satisfiable* if:

$$\exists(x_1, ..., x_n) \in \{0,1\}^n : \varphi(x_1, ..., x_n) = 1$$

Satisfiable (SAT)

$(x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$

$x = 1, y = 0, z = 1$ ✓

Unsatisfiable (UNSAT)

$(x \vee y) \wedge (\neg x \vee y) \wedge (\neg y)$

No assignment works. ✗

Validity (TAUT)

Definition 2: A formula φ is *valid* (a *tautology*) if:

$$\forall (x_1, ..., x_n) \in \{0,1\}^n : \varphi(x_1, ..., x_n) = 1$$

Valid

$(x \vee y) \vee (\neg x \wedge \neg y)$

True for all x, y. ✓

Not valid

$(x \vee y) \wedge (\neg x \vee z)$

False when $x = 0, y = 0$. ✗

SAT vs TAUT

SAT

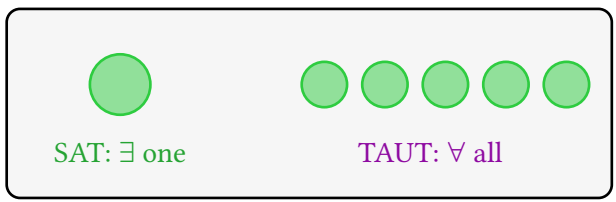
$\exists X : \varphi(X) = 1\,?$

At least one satisfying assignment?

TAUT

$\forall X : \varphi(X) = 1\,?$

All assignments satisfying?



All 2^n assignments

SAT-TAUT Duality

Theorem 1: φ is valid iff $\neg\varphi$ is unsatisfiable.

$$\mathrm{TAUT}(\varphi) \equiv \neg\ \mathrm{SAT}(\neg\varphi)$$

Example: Is $(x \wedge y) \to (x \vee y)$ a tautology?

Check if $\neg((x \wedge y) \to (x \vee y))$ is unsatisfiable:

$$\neg((x \wedge y) \to (x \vee y)) = (x \wedge y) \wedge \neg(x \vee y) = (x \wedge y) \wedge (\neg x \wedge \neg y)$$

Requires $x = 1$ and $x = 0$ simultaneously — contradiction! UNSAT.

Therefore, $(x \wedge y) \to (x \vee y)$ is a tautology. ✓

A SAT solver can check both satisfiability and validity.

Applications

Hardware & Software

- CPU verification (Intel, AMD)
- Circuit equivalence
- Bug finding (Microsoft SAGE)
- Test generation

AI & Optimization

- Planning & scheduling
- Constraint satisfaction
- Combinatorial puzzles
- Configuration problems

Cryptography

- Cipher analysis
- Hash collisions
- Key recovery attacks

Mathematics

- Ramsey theory bounds
- Pythagorean triples
- Graph coloring proofs

SAT Competition (annual since 2002): CaDiCaL, Kissat, Glucose, CryptoMiniSat…

Historical Notes

- **1928:** Hilbert's *Entscheidungsproblem* — can we decide mathematical truth?
- **1936:** Turing & Church: No (for general mathematics).
- **1971:** Cook–Levin theorem: SAT is NP-complete.

For Boolean formulas:

- SAT *is* decidable (try all 2^n assignments)
- Question: can we do better than $O(2^n)$?

P vs NP problem: Is there a polynomial-time algorithm for SAT?

Most believe: **no**. But no proof exists.

CNF: The Language of SAT Solvers

Building Blocks: Literals

Definition 3: A *literal* is a variable or its negation: x or $\neg x$.

Positive literal x

x	value
0	`false`
1	`true`

Negative literal $\neg x$

x	value
0	`true`
1	`false`

The *complement* of literal l is $\bar{l}$: $\overline{x} = \neg x$ $\overline{\neg x} = x$

Building Blocks: Clauses

Definition 4: A *clause* is a disjunction (OR) of literals: $C = (l_1 \vee l_2 \vee \ldots \vee l_k)$

Example: $(x \vee \neg y \vee z)$ — a clause with 3 literals.

A clause is satisfied when *at least one* literal is true.

Here: 7 out of 8 assignments satisfy it!

x	y	z	clause
0	0	0	✓ $(\neg y)$
0	0	1	✓
0	1	0	✗
0	1	1	✓ (z)
1	$*$	$*$	✓ (x)

The only way to falsify a clause: Make *all* its literals false simultaneously.

Conjunctive Normal Form (CNF)

Definition 5: A formula is in **CNF** if it is a conjunction (AND) of clauses:

$$F = C_1 \wedge C_2 \wedge ... \wedge C_m$$

Example:

$$(x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$$

This CNF has 3 clauses over 3 variables.

To satisfy a CNF: Every clause must have at least one true literal.

CNF is unsatisfiable if: Any clause has all literals false.

Special Cases

Unit clause — exactly one literal

(x) forces $x = 1$

$(\neg y)$ forces $y = 0$

No choice! Must satisfy it.

Empty clause $\square$ — no literals

Cannot be satisfied!

$\square$ in formula $\Rightarrow$ **UNSAT**

Contradiction detected.

Example: Formula: $(x \vee y) \wedge (\neg x) \wedge (y \vee z)$

Unit clause $(\neg x)$ forces $x = 0$. After simplification:

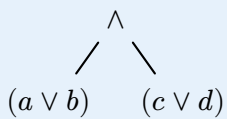
- $(x \vee y)$ becomes (y) — another unit clause!
- Continue: $y = 1$ satisfies everything.

Unit Propagation: Repeatedly apply forced assignments until no unit clauses remain.

CNF vs DNF

CNF (AND of ORs)

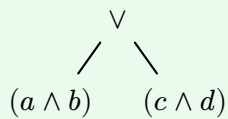
$$\underbrace{(a \vee b)}_{\text{clause}} \wedge \underbrace{(c \vee d)}_{\text{clause}}$$



SAT: **hard** TAUT: **easy**

DNF (OR of ANDs)

$$\underbrace{(a \wedge b)}_{\text{cube}} \vee \underbrace{(c \wedge d)}_{\text{cube}}$$



SAT: **easy** TAUT: **hard**

The key insight:

- CNF-SAT: Must satisfy *all* clauses ⇒ hard global constraint
- DNF-SAT: Just find *one* satisfiable cube ⇒ easy local check

The Conversion Problem

Problem: How do we convert an arbitrary formula to CNF?

Naive conversion can explode!

$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \ldots \vee (x_n \wedge y_n)$

Direct CNF: 2^n clauses (exponential blowup!)

Example: $(a \wedge b) \vee (c \wedge d)$ converts to:

$$(a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$$

4 clauses for 2 cubes. With n cubes: 2^n clauses, each of size n.

We need a smarter approach…

Tseitin Transformation

Definition 6: Tseitin transformation is a method to convert any formula φ into an *equisatisfiable* CNF formula F of size $O(|\varphi|)$ (instead of exponential) by introducing *auxiliary variables* for subformulas.

Idea: For each subformula ψ, introduce a fresh variable t and encode $t \leftrightarrow \psi$.

Example: Formula: $(a \wedge b) \vee c$

Introduce t for $(a \wedge b)$:

- $t \leftrightarrow (a \wedge b)$ encodes as: $(\neg t \vee a) \wedge (\neg t \vee b) \wedge (t \vee \neg a \vee \neg b)$

Final CNF: $(t \vee c) \wedge (\neg t \vee a) \wedge (\neg t \vee b) \wedge (t \vee \neg a \vee \neg b)$

Key: Tseitin CNF is *equisatisfiable* (same SAT/UNSAT status), not equivalent.

That's all we need for SAT solving!

Tseitin Encodings for Gates

Gate	Meaning	CNF clauses
$t \leftrightarrow \neg a$	$t = \neg a$	$(t \vee a) \wedge (\neg t \vee \neg a)$
$t \leftrightarrow a \wedge b$	$t = a \wedge b$	$(\neg t \vee a) \wedge (\neg t \vee b) \wedge (t \vee \neg a \vee \neg b)$
$t \leftrightarrow a \vee b$	$t = a \vee b$	$(t \vee \neg a) \wedge (t \vee \neg b) \wedge (\neg t \vee a \vee b)$
$t \leftrightarrow a \rightarrow b$	$t = \neg a \vee b$	$(t \vee a) \wedge (t \vee \neg b) \wedge (\neg t \vee \neg a \vee b)$
$t \leftrightarrow a \leftrightarrow b$	$t = (a \equiv b)$	$(\neg t \vee \neg a \vee b) \wedge (\neg t \vee a \vee \neg b) \wedge \ldots$

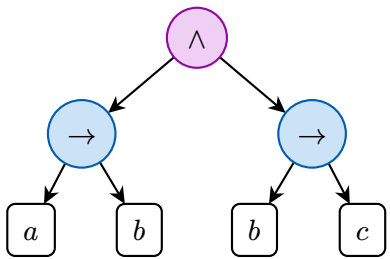
Each gate adds a constant number of clauses.

Total CNF size: $O(|\varphi|)$ — linear in formula size!

Tseitin Transformation: Step-by-Step

Example: Convert $\varphi = (a \to b) \wedge (b \to c)$ to CNF using Tseitin transformation.

Step 1: Build the formula tree.



Step 2: Introduce auxiliary variables.

- $t_1 \leftrightarrow (a \to b)$
- $t_2 \leftrightarrow (b \to c)$
- $t_3 \leftrightarrow (t_1 \wedge t_2)$ *(root)*

Step 3: Encode each gate.

$t_1 \leftrightarrow (a \to b)$: $(t_1 \vee a)(t_1 \vee \neg b)(\neg t_1 \vee \neg a \vee b)$

$t_2 \leftrightarrow (b \to c)$: $(t_2 \vee b)(t_2 \vee \neg c)(\neg t_2 \vee \neg b \vee c)$

$t_3 \leftrightarrow (t_1 \wedge t_2)$: $(\neg t_3 \vee t_1)(\neg t_3 \vee t_2)(t_3 \vee \neg t_1 \vee \neg t_2)$

Step 4: Assert root is true: (t_3)

Why SAT is Hard: Conflicting Constraints

The challenge: Each clause is easy to satisfy alone, but clauses *interact* through shared variables.

Example: $(x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$

- C1 wants x or y true
- C2 wants $\neg x$ or z true
- C3 wants $\neg y$ or $\neg z$ true

Setting $x = 1$ satisfies C1, but now C2 needs $z = 1$.

Setting $z = 1$ means C3 needs $\neg y$, so $y = 0$.

Check: $x = 1, y = 0, z = 1$ — all satisfied! ✓

Example: $(x) \wedge (\neg x \vee y) \wedge (\neg y)$

- C1 forces $x = 1$
- C2 with $x = 1$ forces $y = 1$
- C3 forces $y = 0$

Contradiction! ✗

The constraints propagate and eventually clash.

SAT solvers exploit this: propagate forced assignments, detect conflicts early.

Models and Satisfying Assignments

Definition 7: A *model* of a formula φ is an assignment that makes φ true.

Notation: $\sigma \vDash \varphi$ means "σ is a model of φ" or "σ satisfies φ".

Example: $\varphi = (x \vee y) \wedge (\neg x \vee z)$

$\sigma_1 = \{x = 1, y = 0, z = 1\}$: $\sigma_1 \vDash \varphi$ ✓

$\sigma_2 = \{x = 1, y = 0, z = 0\}$: $\sigma_2 \nvDash \varphi$ ✗

Terminology:

- "model" = "satisfying assignment"
- φ is SAT iff φ has a model
- φ is UNSAT iff φ has no models

Broader context: In logic, a *model* is a structure that makes a set of formulas true.

- For propositional logic: model = truth assignment.
- For first-order logic: model = domain + interpretation of symbols.

The study of models is called *model theory* — a major branch of mathematical logic.

Decision vs Search

Decision SAT

Input: CNF formula F

Output: SAT or UNSAT

"Does F have a model?"

Search SAT (Functional SAT)

Input: Satisfiable CNF formula F

Output: A model $\sigma \vDash F$

"Find me a model!"

Surprising fact: These problems are *equally hard* (both NP-complete).

If you can decide SAT in polynomial time, you can also *find* models in polynomial time (by self-reduction).

Example: $F = (x \vee y) \wedge (\neg x \vee y) \wedge (\neg y \vee z)$ Decision: **SAT** Model: $\sigma = \{x = 0, y = 1, z = 1\} \vDash F$

Modern solvers return both: `SAT` with a model, or `UNSAT` (sometimes with a proof).

Complexity of SAT

The Brute Force Approach

Naive algorithm: Try all possible assignments.

- For n variables: 2^n possible assignments
- Check each assignment in $O(m)$ time (where m = number of clauses)
- Total: $O(m \cdot 2^n)$ — exponential!

Example:

- $n = 10$: 1024 assignments (instant)
- $n = 20$: $\approx 10^6$ assignments (seconds)
- $n = 50$: $\approx 10^{15}$ assignments (years!)
- $n = 100$: $\approx 10^{30}$ assignments (heat death of universe)

Key question: Is there a polynomial-time algorithm for SAT?

This is the famous **P vs NP** problem — worth $1 million!

Complexity Classes: P and NP

Definition 8: The *complexity class* P consists of problems solvable in polynomial time by a deterministic algorithm.

Examples: sorting, shortest path, linear programming.

Definition 9: The *complexity class* NP consists of problems where a "yes" answer can be *verified* in polynomial time.

Equivalently: solvable in polynomial time by a *nondeterministic* algorithm.

SAT is in NP: Given a satisfying assignment (certificate), we can verify it in $O(n + m)$ time by evaluating each clause.

NP-Completeness

Definition 10: A problem X is *NP-hard* if every problem in NP can be *reduced* to X in polynomial time.

Definition 11: A problem is *NP-complete* if it is both in NP and NP-hard.

NP-complete problems are the "hardest" problems in NP.

If you solve ANY NP-complete problem in polynomial time, you solve ALL of them!

This would prove P = NP and revolutionize computer science.

The Cook–Levin Theorem

Theorem 2 (Cook–Levin (1971)): SAT is NP-complete.

That is, SAT is in NP, and *any* problem in NP can be *reduced* to SAT in polynomial time.

Historical significance:

- First problem proven NP-complete
- Independently discovered by Stephen Cook (1971) and Leonid Levin (1973)
- Foundation of computational complexity theory

Consequence: If we could solve SAT in polynomial time, we could solve:

- Traveling salesman, graph coloring, scheduling, ...
- Cryptographic problems, protein folding, ...
- Essentially all "hard" combinatorial problems!

Proof Sketch: SAT is NP-complete

Proof *(SAT ∈ NP)*: A satisfying assignment serves as a *certificate* that can be verified in linear time by evaluating each clause. □

Proof *(SAT is NP-hard, sketch)*: For any problem $L \in \text{NP}$, there exists a polynomial-time verifier V.

Given input x, we construct a formula φ_x such that:

- φ_x encodes the computation of V on x with certificate c
- Variables represent: machine state, tape contents, head position at each step
- Clauses enforce: valid initial state, transition rules, acceptance

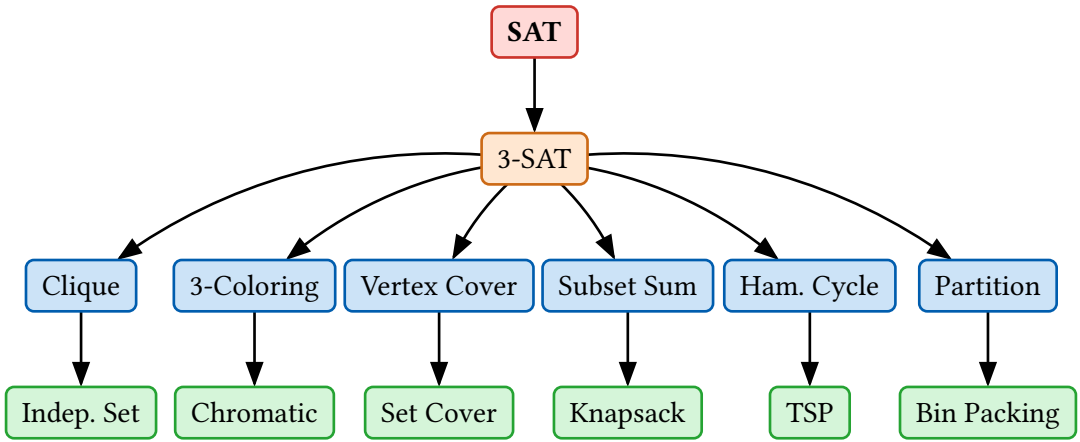
Then: $x \in L$ iff φ_x is satisfiable.

Construction is polynomial in $|x|$. □

This result means SAT is a "universal" problem — all of NP reduces to it.

Karp's 21 NP-Complete Problems (1972)

Richard Karp showed 21 classic problems are NP-complete by reducing SAT to them.



To prove that problem X is NP-complete: *reduce* a known NP-complete problem (*e.g.*, 3-SAT) to X.

SAT Variants: The Complexity Landscape

Definition 12: *k-SAT* is SAT restricted to formulas where each clause has exactly k literals.

In P (polynomial time):

- **1-SAT:** Trivial (unit propagation)
- **2-SAT:** $O(n + m)$ via implication graphs
- **Horn-SAT:** At most one positive literal per clause — $O(n \cdot m)$
- **XOR-SAT:** Gaussian elimination — $O(n^3)$

NP-complete:

- **3-SAT:** The "minimal" hard case
- **k-SAT** for all $k \geq 3$
- **NAE-SAT:** Not-all-equal SAT
- **1-in-3 SAT:** Exactly one true per clause

The phase transition: Going from 2-SAT to 3-SAT crosses the P/NP boundary!

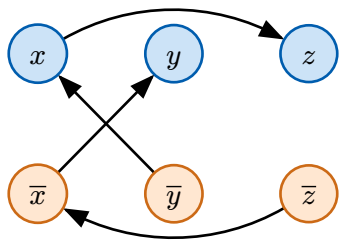
This is one of the sharpest complexity transitions known.

2-SAT: A Polynomial Algorithm

Definition 13: The *implication graph* for a 2-SAT formula φ is a directed graph G_φ where:

- Nodes: literals x and $\neg x$ for each variable
- Edges: $(\neg a, b)$ and $(\neg b, a)$ for each clause $(a \vee b)$

Example: Formula: $(x \vee y) \wedge (\neg x \vee z)$



Clause $(a \vee b)$ becomes implications $\neg a \to b$ and $\neg b \to a$.

2-SAT: A Polynomial Algorithm [2]

Theorem 3 (2-SAT Characterization): A 2-SAT formula is unsatisfiable iff there exists a variable x such that x and $\neg x$ are in the same strongly connected component of G_φ.

Algorithm: Tarjan's SCC in $O(n + m) \Rightarrow$ 2-SAT in linear time!

SAT Solving Algorithms

How to Solve SAT?

Despite exponential worst-case complexity, modern SAT solvers are remarkably effective:

- Handle industrial instances with millions of variables
- Often find solutions in seconds or minutes
- Based on *clever search strategies* and *learning*

Key techniques:

- Unit propagation
- Pure literal elimination
- Backtracking search (DPLL)
- Conflict-driven clause learning (CDCL)
- Variable selection heuristics

Modern solvers:

- MiniSat, CryptoMiniSat
- Glucose, Lingeling
- CaDiCaL, Kissat
- Z3, CVC5 (SMT solvers)

Unit Propagation

Definition 14: *Unit propagation* is the following simplification rule: if a CNF formula contains a *unit clause* (l), then:

1. Set the literal l to true
2. Remove all clauses containing l (they are satisfied)
3. Remove $\bar{l}$ from all remaining clauses

Example: Formula: $(x \vee y) \wedge (\neg x \vee z) \wedge (x)$

Unit clause (x) forces $x = 1$:

- Remove $(x \vee y)$ and (x), since they contain x
- Remove $\neg x$ from $(\neg x \vee z)$

Result: (z) — another unit clause!

Continue: $z = 1$, formula simplifies to $\top$, *i.e.*, the formula is *satisfied*.

Unit propagation is the *workhorse* of SAT solvers — fast and effective!

Pure Literal Elimination

Definition 15: A literal l is *pure* if it appears in the formula but $\bar{l}$ does not.

Definition 16: The *pure literal rule* states: if a literal l is pure, then:

1. Set l to true
2. Remove all clauses containing l

Example: Formula: $(x \vee y) \wedge (x \vee z) \wedge (y \vee z)$

The literal x is pure (only positive occurrences).

Set $x = 1$, remove clauses containing x:

Result: $(y \vee z)$ — simpler formula!

Pure literals can always be set to true without losing satisfiability.

The DPLL Algorithm

Definition 17: *DPLL* (Davis–Putnam–Logemann–Loveland, 1962) is a complete backtracking algorithm for SAT:

1. **Simplify:** Apply unit propagation and pure literal elimination
2. **Check:** If no clauses remain, return SAT; if empty clause exists, return UNSAT
3. **Branch:** Choose an unassigned variable x
   - Try $x = 1$; if SAT, return SAT
   - Otherwise, try $x = 0$; return result

Key properties:

- *Complete:* always finds a solution if one exists
- *Sound:* only returns SAT if formula is satisfiable
- Worst case: exponential time (unavoidable unless P = NP)

DPLL: Visual Example

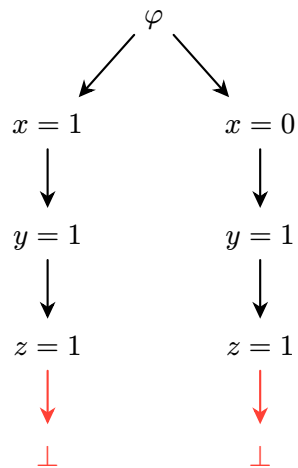
Example:

Formula: $(x \vee y) \wedge (\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg z)$

DPLL explores:

- Branch $x = 1$: unit prop gives $y = 1$, then $z = 1$ — conflict with $(\neg z)$!
- Backtrack, try $x = 0$: unit prop gives $y = 1$, then $z = 1$ — same conflict!

Both branches lead to conflict $\Rightarrow$ **UNSAT**.

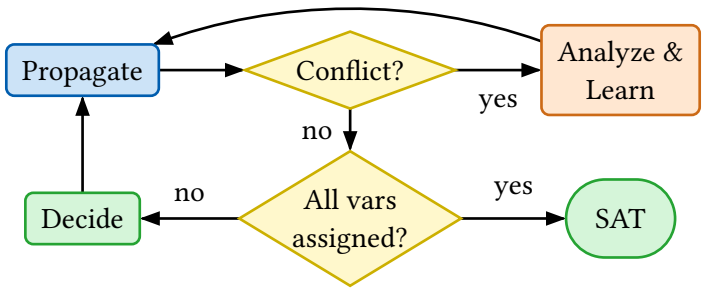


Conflict-Driven Clause Learning (CDCL)

Modern SAT solvers use CDCL — an enhanced version of DPLL.

Key insight: When we hit a conflict, *learn from it*!

Analyze *why* the conflict occurred, and add a new clause to prevent repeating the same "mistake".

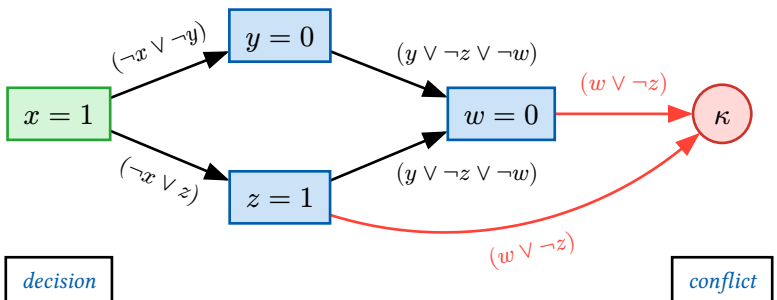


CDCL can *skip* large parts of the search space by learning conflict clauses.

Implication Graphs & Conflict Analysis

Definition 18: An *implication graph* is a DAG tracking why each literal was assigned:

- Decision nodes: literals chosen by branching
- Propagation nodes: literals forced by unit propagation
- Edges: from antecedent literals to implied literal



Implication Graphs & Conflict Analysis [2]

Clauses correspond to implications, visualized as edges in the graph:

Clause	Implication
$(\neg x \vee \neg y)$	$x \to \neg y$
$(\neg x \vee z)$	$x \to z$
$(y \vee \neg z \vee \neg w)$	$(y \wedge z) \to \neg w$
$(w \vee \neg z)$	$(\neg w \wedge z) \to \bot$

Definition 19: The *first UIP* (unique implication point) scheme derives the conflict clause by finding a cut in the implication graph that separates the conflict from the last decision.

The **1st UIP** finds the closest such cut — most effective in practice!

Non-chronological backtracking: Jump back to the decision level of the *second-highest* literal in the learned clause — can skip many levels!

Alternative Learning Schemes

1st UIP is standard, but alternatives exist:

- **All-UIP:** Learn all UIPs at current level
- **Last UIP (Decision):** Learn clause containing only decision variables
- **RelSAT scheme:** Learn clause of minimum size

Theorem 4 (1st UIP Optimality): 1st UIP produces *shortest* asserting clause at the current level.

Empirical result: 1st UIP dominates in practice, but combining different learning schemas can help on specific instances.

Lookahead Solvers

Definition 20: A *lookahead solver* probes each variable before branching:

- Temporarily set $x = 0$, propagate, measure effects
- Temporarily set $x = 1$, propagate, measure effects
- Choose variable that maximizes some criterion

Lookahead benefits:

- Detects failed literals (probing gives conflict)
- Learns *necessary assignments*
- Computes better branching heuristic

Lookahead drawbacks:

- Expensive: $O(n)$ propagations per decision
- Doesn't learn clauses as effectively as CDCL

Best of both worlds: Use lookahead for cube generation, CDCL for solving — *cube-and-conquer*.

Phase Saving

Definition 21: *Phase saving* is a heuristic that remembers the last assignment of each variable.

When branching on x, first try x's previous value.

Why it helps:

- Good assignments tend to remain good after restarts
- Avoids re-exploring same wrong choices
- Very cheap to implement (just remember last value)

Combined with restarts: Phase saving + aggressive restarts = very effective!

Solver can restart frequently while preserving good partial solutions.

The Two-Watched Literals Scheme

The bottleneck: Unit propagation happens millions of times — must be FAST!

Definition 22: The *two-watched literals* scheme maintains pointers to exactly two unassigned literals ("watches") for each clause.

Key property: Only check a clause when one of its watched literals becomes false.

Why it works:

- If both watches are unassigned or true ⇒ clause is not unit
- When a watch becomes false, search for a new unassigned literal
- If none found and other watch is unassigned ⇒ unit clause!
- If none found and other watch is false ⇒ conflict!

Impact: Chaff (2001) introduced this, achieving 10–100× speedup!

Two-watched literals is the *standard* in all modern SAT solvers.

Variable Selection Heuristics

VSIDS (Variable State Independent Decaying Sum):

- Each variable has an *activity score*
- Increment score when variable appears in a conflict
- Periodically *decay* all scores (multiply by 0.95)
- Always branch on the highest-scoring variable

Why VSIDS works:

- Recently conflicting variables are "hot" — likely relevant
- Decay ensures we don't get stuck on old conflicts
- Focuses search on the "active" part of the formula

Modern variants:

- **CHB** (Conflict History-Based): learning rate adaptation
- **LRB** (Learning Rate Branching): exponential moving average
- **VMTF** (Variable Move To Front): simpler, sometimes effective

Restarts: The Counter-Intuitive Trick

Observation: Sometimes CDCL gets stuck in a "bad" part of the search space.

Solution: Periodically *restart* the search from scratch!

Definition 23: A *restart policy* specifies when to restart the search: after k conflicts, undo all decisions but keep learned clauses.

Common policies:

- **Luby sequence:** 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, ... (optimal for Las Vegas algorithms)
- **Geometric:** $k, k \cdot r, k \cdot r^2, \ldots$ (*e.g.*, $k = 100$, $r = 1.5$)
- **Glucose-style:** Restart when recent learned clauses have high LBD

Why restarts help:

- Escape from bad variable orderings
- Learned clauses persist $\Rightarrow$ progress is not lost
- Combines exploration with exploitation

Encoding Problems as SAT

The SAT Encoding Recipe

General approach to solve any problem with SAT:

1. **Variables:** Define propositional variables to represent the problem's state
2. **Constraints:** Encode requirements as Boolean formulas
3. **CNF:** Convert to conjunctive normal form
4. **Solve:** Run a SAT solver
5. **Decode:** Interpret the solution (if SAT)

Example: **Problem:** Schedule 3 tasks without conflicts

Variables: $t_{i,s}$ = "task i runs in slot s"

Constraints:

- Each task runs in some slot: $(t_{1,1} \vee t_{1,2}) \wedge \ldots$
- No two tasks in same slot: $(\neg t_{1,1} \vee \neg t_{2,1}) \wedge \ldots$

Example: Graph Coloring

Definition 24: A *graph k-coloring* is an assignment of one of k colors to each vertex of a graph $G = (V, E)$ such that adjacent vertices have different colors.

SAT encoding:

Variables: $x_{v,c}$ = "vertex v has color c"

Constraints:

- Each vertex has at least one color: $\bigvee_c x_{v,c}$ for each v
- Each vertex has at most one color: $\left(\neg x_{v,c_1} \vee \neg x_{v,c_2}\right)$ for $c_1 \neq c_2$
- Adjacent vertices differ: $\left(\neg x_{u,c} \vee \neg x_{v,c}\right)$ for each edge (u, v)

Example: Triangle with 3 colors: 9 variables, 24 clauses.

SAT solver finds coloring instantly!

Example: Sudoku as SAT

A standard 9×9 Sudoku puzzle can be encoded as a SAT instance:

Variables: $x_{r,c,d}$ = "cell (r, c) contains digit d" — 729 variables

Constraints:

- Each cell has exactly one digit (81 cells × 36 clauses)
- Each row has all digits (9 rows × 9 digits × 36 clauses)
- Each column has all digits (9 columns × 9 digits × 36 clauses)
- Each 3×3 box has all digits (9 boxes × 9 digits × 36 clauses)
- Given clues (unit clauses)

Result: $\approx$ 12,000 clauses total.

Modern SAT solvers solve any Sudoku in *milliseconds*!

Complete SAT Workflow Example

Problem: Schedule 3 tasks on 2 time slots with constraints.

Tasks: A, B, C. Slots: 1, 2. Constraint: A and B cannot be in the same slot.

Step 1: Variables — $t_{X,s}$ = "task X in slot s"

	Slot 1	**Slot 2**
A	$t_{A,1}$	$t_{A,2}$
B	$t_{B,1}$	$t_{B,2}$
C	$t_{C,1}$	$t_{C,2}$

Step 2: Clauses

- Each task in some slot: $(t_{A,1} \vee t_{A,2})$, $(t_{B,1} \vee t_{B,2})$, $(t_{C,1} \vee t_{C,2})$
- Conflict constraint: $(\neg t_{A,1} \vee \neg t_{B,1})$ and $(\neg t_{A,2} \vee \neg t_{B,2})$

Step 3: Solve — SAT! Model: $t_{A,1} = 1, t_{B,2} = 1, t_{C,1} = 1$

Step 4: Decode — A in slot 1, B in slot 2, C in slot 1 ✓

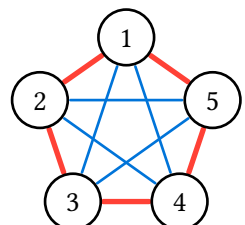
Example: Ramsey Numbers

Definition 25 (Ramsey Number): The *Ramsey number* $R(r, s)$ is the smallest n such that any 2-coloring of edges of K_n contains either a red K_r or a blue K_s.

Example (Finding $R(3,3) = 6$): Can we 2-color edges of K_5 without monochromatic triangles?

Encoding:

- Variables: $e_{i,j}$ = "edge (i, j) is red"
- For each triangle (i, j, k):
  - Not all red: $(\neg e_{i,j} \vee \neg e_{j,k} \vee \neg e_{i,k})$
  - Not all blue: $(e_{i,j} \vee e_{j,k} \vee e_{i,k})$



SAT solver: **SAT** for K_5, **UNSAT** for K_6.

Therefore $R(3, 3) = 6$.

SAT Encoding: At-Most-One Constraint

A common constraint: "at most one of $x_1, ..., x_n$ is true."

Pairwise encoding:

$$\bigwedge_{1 \le i < j \le n} (\neg x_i \vee \neg x_j)$$

This generates $\binom{n}{2} = \frac{n(n-1)}{2}$ clauses.

Example: For $n = 4$ variables: 6 clauses.

$$(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_4)$$

Warning: For large n, this encoding grows quadratically. Advanced encodings (commander, ladder) use $O(n)$ clauses.

SAT Encoding: Exactly-One Constraint

Definition 26: An *exactly-one constraint* requires that exactly one of $x_1, \ldots, x_n$ is true.

Encoding:

At least one: $(x_1 \vee x_2 \vee \ldots \vee x_n)$ — 1 clause

At most one: $(\neg x_i \vee \neg x_j)$ for all $i < j$ — $\binom{n}{2}$ clauses

Example: For task scheduling: "each task runs in exactly one time slot."

Variables: $t_{i,1}, t_{i,2}, t_{i,3}$ for task i in slots 1, 2, 3.

Exactly-one: $(t_{i,1} \vee t_{i,2} \vee t_{i,3}) \wedge (\neg t_{i,1} \vee \neg t_{i,2}) \wedge (\neg t_{i,1} \vee \neg t_{i,3}) \wedge (\neg t_{i,2} \vee \neg t_{i,3})$

Advanced Encodings

XOR Constraints and Gaussian Elimination

Definition 27: An *XOR clause* is a constraint requiring an odd number of literals to be true:

$$x_1 \oplus x_2 \oplus ... \oplus x_n = 1$$

Problem: XOR of n variables requires 2^{n-1} CNF clauses!

$(x \oplus y \oplus z)$ becomes $(x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$

Solution: Native XOR support

CryptoMiniSat and other solvers handle XOR constraints *natively*:

- Gaussian elimination over GF(2)
- Linear algebra instead of search
- Polynomial time for pure XOR systems

XOR Constraints and Gaussian Elimination [2]

Example: Cryptographic problems (AES, SHA) have many XOR constraints.

CryptoMiniSat: 100× faster than standard SAT on crypto instances!

Parity Reasoning

Definition 28: *XOR-SAT* is SAT restricted to XOR clauses only.

Solvable in polynomial time via Gaussian elimination!

Hybrid solving:
- Maintain CNF and XOR parts separately
- Propagate assignments between them
- Use Gaussian elimination for XOR part

Applications:
- Cryptanalysis (breaking ciphers)
- Error-correcting codes (decoding)
- Randomness testing (analyzing sequences)

Theorem 5 (XOR Reasoning Power): Resolution cannot efficiently simulate XOR reasoning.

Some formulas need exponential resolution proofs but have polynomial XOR proofs.

Cardinality Constraints

Definition 29: A *cardinality constraint* restricts how many of $x_1, \ldots, x_n$ can be true:

- **At-most-k:** $\sum_{i=1}^{n} x_i \leq k$
- **At-least-k:** $\sum_{i=1}^{n} x_i \geq k$
- **Exactly-k:** $\sum_{i=1}^{n} x_i = k$

Challenge: Naive pairwise encoding for at-most-k requires $O(n^k)$ clauses!

For $n = 100$, $k = 3$: over 160,000 clauses.

Better encodings:

- **Sequential counter:** $O(n \cdot k)$ clauses and variables
- **Parallel counter:** $O(n \cdot \log k)$ clauses
- **Sorting networks:** $O(n \cdot \log^2 n)$ clauses
- **BDD-based:** Often optimal in practice

Choice of encoding dramatically affects solver performance!

Sequential Counter Encoding

Definition 30: The *sequential counter* encoding (Sinz, 2005) introduces auxiliary variables $s_{i,j}$ meaning "at least j of $x_1, \ldots, x_i$ are true."

	x_1	x_2	x_3	x_4
count ≥ 1	$s_{1,1}$	$s_{2,1}$	$s_{3,1}$	$s_{4,1}$
count ≥ 2		$s_{2,2}$	$s_{3,2}$	$s_{4,2}$

Advantage: UP achieves *arc consistency*

Key clauses:

- $x_i \rightarrow s_{i,1}$: input contributes to count
- $s_{i-1,j} \rightarrow s_{i,j}$: propagate count forward
- $x_i \wedge s_{i-1,j-1} \rightarrow s_{i,j}$: increment when both true
- $\neg s_{n,k+1}$: forbid exceeding limit (for at-most-k)

Example: At-most-2 constraint: add clause $\neg s_{4,3}$ (count never reaches 3).

If $x_1 = x_2 = x_3 = 1$, propagation sets $s_{3,3} = 1 \Rightarrow$ conflict!

Arithmetic in SAT: Bit-Blasting

Definition 31: *Bit-blasting* encodes integer variables as bit-vectors and arithmetic operations as Boolean circuits.

Example (Integer addition): $z = x + y$ where x, y, z are 4-bit integers.

Variables: x_0, x_1, x_2, x_3 (bits of x), similarly for y, z.

Encode using full adders:

- $z_i = x_i \oplus y_i \oplus c_{i-1}$ (sum bit)
- $c_i = (x_i \wedge y_i) \vee (c_{i-1} \wedge (x_i \oplus y_i))$ (carry)

Applications:

- Software verification (bounded model checking)
- Cryptographic analysis
- SMT bit-vector theory (via bit-blasting to SAT)

Pseudo-Boolean Constraints

Definition 32: A *pseudo-Boolean (PB) constraint* is a linear inequality over Boolean variables:

$$\sum_{i=1}^{n} a_i \cdot x_i \leq k \quad \text{where } a_i, k \in \mathbb{Z}$$

Example: $3x_1 + 2x_2 + x_3 + x_4 \leq 4$

Allows at most: x_1 alone, or $x_2 + x_3 + x_4$, or $x_2 + x_3$, *etc.*

Encodings:

- **Adder networks:** Build binary addition circuit
- **BDD-based:** Construct BDD for the constraint, convert to CNF
- **Sorting networks:** For unweighted or low-weight cases
- **Watchdog encoding:** Generalization of sequential counter

PB constraints are powerful for optimization problems and occur frequently in MaxSAT.

Incremental SAT and Assumptions

Incremental SAT Solving

Definition 33: *Incremental SAT* is the task of solving a sequence of related SAT problems, reusing information from previous solves.

Common scenarios:

- Bounded model checking: $\varphi_0, \varphi_0 \wedge \varphi_1, \varphi_0 \wedge \varphi_1 \wedge \varphi_2, \ldots$
- Iterative refinement: add constraints until satisfied
- AllSAT: repeatedly block found solutions
- CEGIS: add counterexamples

Key insight: Learned clauses from previous solves remain valid!

Reusing learned clauses can give $10\times$ – $100\times$ speedup.

Assumption-Based Solving

Definition 34: *Assumptions* are temporary unit clauses that can be "retracted" between solves.

`solve(assumptions = [x, ¬y])` acts like adding (x) and $(\neg y)$, but only for this call.

API (MiniSat-style):

```
solver.add_clause([x, y, z])       // Permanent
solver.solve([a, ¬b])              // Temporary assumptions
solver.solve([¬a, b])              // Different assumptions, same clauses
```

Example: **Checking multiple properties:**

Base formula: φ (system behavior)

Check property 1: `solve([¬property1])` — if UNSAT, property holds

Check property 2: `solve([¬property2])` — same base formula!

Unsat Cores via Assumptions

Definition 35: An *unsatisfiable core* is a subset of clauses that is still unsatisfiable.

If `solve(assumptions = [a1, a2, a3, ...])` returns UNSAT, solver can report which assumptions are in the conflict.

These form an *unsatisfiable core* — subset of assumptions sufficient for UNSAT.

Example: **Debugging infeasible constraints:**

```
solve([constraint1, constraint2, constraint3, constraint4])
```

Returns UNSAT with core: `{constraint1, constraint3}`

⇒ Constraints 1 and 3 are mutually incompatible!

Applications: MaxSAT (core-guided), diagnosis, debugging, minimal explanations.

Push/Pop Interface

Definition 36: A *context stack* allows adding and removing clauses in a stack-like manner:

- `push()`: Save current state
- `pop()`: Restore to last pushed state, removing added clauses

Example:

```
solver.add(base_clauses)
solver.push()
solver.add(additional_clauses)
result1 = solver.solve()
solver.pop()  // Removes additional_clauses
solver.push()
solver.add(other_clauses)
result2 = solver.solve()
```

Limitation: Less efficient than assumptions — learned clauses may become invalid.

Use assumptions when possible, push/pop for clause structure changes.

SMT: Advanced Topics

Theory Combination: Nelson-Oppen

Problem: SMT formula uses *multiple* theories simultaneously.

Example: $(x = y + 1) \wedge (f(x) \neq f(y + 1))$

Uses: Linear arithmetic (LIA) + Uninterpreted functions (UF)

Definition 37: The *Nelson-Oppen method* combines theory solvers for *disjoint signature* theories:

1. Purify: separate theory-specific parts
2. Share equalities between shared variables
3. Iterate until fixed point or conflict

Example: $(x = y + 1) \wedge (f(x) \neq f(y + 1))$

LIA sees: $x = y + 1$

UF sees: $f(x) \neq f(y + 1)$

Theory Combination: Nelson-Oppen [2]

LIA deduces: $x = y + 1$, shares with UF

UF deduces: conflict with $f(x) \neq f(y + 1)$ since arguments equal!

Theory Propagation and Conflict

DPLL(T) communication:

- **SAT → Theory:** "Here's a partial assignment, is it theory-consistent?"
- **Theory → SAT:** "Conflict!" (theory lemma) or "These literals are implied"

Definition 38: *Theory propagation* is when the theory solver deduces new facts from assigned literals.

Example: From $x \leq y$ and $y \leq z$, deduce $x \leq z$.

Definition 39: A *theory conflict* occurs when the theory solver finds that an assignment is theory-inconsistent.

It returns a *theory lemma* explaining the conflict.

Example: Assignment: $x > 5, y < 3, x < y$

Theory Propagation and Conflict [2]

LIA theory: Conflict! $x > 5$ and $x < y$ and $y < 3$ is impossible.

Lemma: $(x \leq 5) \vee (y \geq 3) \vee (x \geq y)$

Lazy vs Eager SMT

Lazy approach (DPLL(T)):

- Boolean skeleton to SAT solver
- Theory solver checks consistency
- Exchange lemmas on-demand

Pros: Flexible, handles complex theories

Cons: Many theory calls

Eager approach (bit-blasting):

- Compile entire formula to SAT
- Run pure SAT solver
- Decode model back

Pros: Simple, SAT solver optimizations

Cons: Huge formulas, loses structure

Modern SMT: Hybrid approaches — eager for bit-vectors, lazy for arithmetic/arrays.

SMT Theories: Details

Definition 40: The *theory of arrays* provides:

- **Read:** $\text{read}(a, i)$ returns element at index i
- **Write:** $\text{write}(a, i, v)$ returns array with $a[i] = v$

Axioms:

- $\text{read}(\text{write}(a, i, v), i) = v$
- $i \neq j \rightarrow \text{read}(\text{write}(a, i, v), j) = \text{read}(a, j)$

Definition 41: The *theory of strings* includes:

- Concatenation: $x \cdot y$
- Length: $|x|$
- Contains: $x \in y$
- Regular expressions: $x \in L(r)$

Challenge: Undecidable in general! Solvers use heuristics.

SMT Theories: Details [2]

Theory support varies by solver:

- Z3: Most complete theory support
- CVC5: Strong on strings and arithmetic
- Yices: Fast on linear arithmetic

Parameterized Complexity of SAT

Fixed-Parameter Tractability

Definition 42: A *parameterized problem* is an instance (x, k) where x is the input and k is a *parameter*.

Definition 43: A problem is *fixed-parameter tractable (FPT)* if it is solvable in time $f(k) \cdot |x|^{O(1)}$ for some computable f.

Exponential in parameter k, but *polynomial* in input size!

Key insight: If k is small, FPT algorithms can be practical even when $f(k)$ is large.

FPT Results for SAT

Theorem 6 (Bounded Treewidth): SAT restricted to CNF formulas with *treewidth k* is solvable in $O(2^k \cdot n)$ time.

Treewidth: Measures how "tree-like" the variable interaction graph is.

- Tree: treewidth 1
- Grid: treewidth $\sqrt{n}$
- Complete graph: treewidth n

Theorem 7 (Backdoor Variables): If there exist k variables whose removal makes the formula tractable (*e.g.*, Horn), SAT is FPT parameterized by k.

Practical impact: Many industrial formulas have small treewidth or backdoors!

FPT Results for SAT [2]

This partially explains why SAT solvers work well despite NP-hardness.

Kernelization

Definition 44: *Kernelization* is a polynomial-time reduction $(x, k) \to (x', k')$ where $|x'|, k' \leq f(k)$.

It produces a *kernel* of size depending only on the parameter.

Negative result: k-SAT (parameterized by number of clauses) has no polynomial kernel unless NP $\subseteq$ coNP/poly.

This is unlikely, so SAT probably cannot be efficiently compressed!

Positive result: SAT parameterized by *deletion distance to Horn* has a polynomial kernel.

If formula is "almost Horn," we can reduce it efficiently.

Extensions and Variants

MaxSAT: Optimization over SAT

Definition 45: *MaxSAT* is the optimization problem: given a CNF formula, find an assignment that *maximizes* the number of satisfied clauses.

Definition 46: *Weighted MaxSAT* assigns a weight to each clause; the goal is to maximize the total weight of satisfied clauses.

Definition 47: *Partial MaxSAT* distinguishes two types of clauses:
- **Hard clauses:** Must be satisfied (weight = ∞)
- **Soft clauses:** Want to satisfy, but not required

Example: Scheduling with preferences:
- Hard: "No two meetings at same time"
- Soft: "Meeting A prefers morning" (weight 5)
- Soft: "Meeting B prefers afternoon" (weight 3)

MaxSAT: Optimization over SAT [2]

Applications: Optimization, diagnosis, repair, configuration, planning.

Solvers: MaxHS, RC2, Open-WBO, EvalMaxSAT

MaxSAT Algorithms

Core-guided approach:

1. Find an unsatisfiable core (subset of soft clauses that conflict)
2. Add cardinality constraint to allow relaxation
3. Iterate until SAT

Branch-and-bound:

- Lower bound: current solution quality
- Upper bound: optimistic estimate
- Prune when bounds cross

Example: Formula: $(x) \wedge (\neg x) \wedge (y)$ (all soft, weight 1)

Core: $\{(x), (\neg x)\}$ — cannot both be satisfied.

Relax: at most 1 of these can be true.

MaxSAT Algorithms [2]

Optimal: satisfy (y) and one of (x), $(\neg x) \Rightarrow$ value = 2.

Quantified Boolean Formulas (QBF)

Definition 48 (QBF): Boolean formulas with *quantifiers*:

$$\exists x_1.\forall x_2.\exists x_3....\varphi(x_1, x_2, x_3, ...)$$

Complexity: QBF satisfiability is **PSPACE-complete** — much harder than NP!

Example: $\exists x.\forall y.(x \vee y) \wedge (\neg x \vee \neg y)$

Does there exist x such that for *all* y, the formula is true?

Try $x = 1$: need $(1 \vee y) \wedge (0 \vee \neg y) = 1 \wedge \neg y$. Fails for $y = 1$.

Try $x = 0$: need $(0 \vee y) \wedge (1 \vee \neg y) = y \wedge 1 = y$. Fails for $y = 0$.

UNSAT — no winning strategy for $\exists$!

Quantified Boolean Formulas (QBF) [2]

Applications: Verification, planning under uncertainty, game solving, synthesis.

Special Cases of SAT

Some SAT restrictions are solvable in polynomial time:

Definition 49 (2-SAT): Each clause has *at most 2 literals*.

Solvable in $O(n + m)$ using implication graphs and strongly connected components!

Definition 50 (Horn-SAT): Each clause has *at most one positive literal*.

Solvable in linear time using unit propagation.

Key insight: The jump from 2-SAT to 3-SAT crosses the P/NP boundary!

3-SAT (each clause has exactly 3 literals) is NP-complete.

The DIMACS Format

Standard input format for SAT solvers:

```
c This is a comment
p cnf 3 4
1 -2 0
2 3 0
-1 -3 0
1 3 0
```

Format:

- `p cnf <vars> <clauses>` — problem line
- Each clause: space-separated integers, terminated by `0`
- Positive integer = positive literal, negative = negated
- Variables numbered $1, 2, 3, \ldots$

Example: The formula $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_3)$

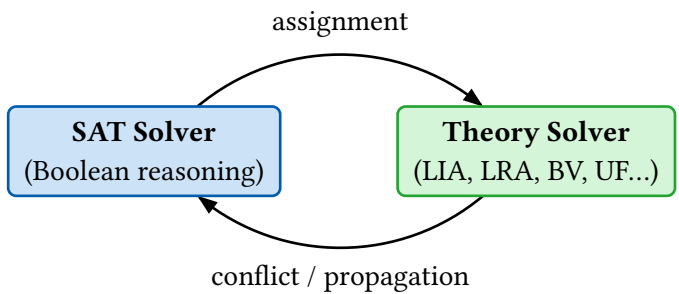
SMT: Satisfiability Modulo Theories

Definition 51 (SMT): Extends propositional SAT with *first-order theories*:

$$\varphi_{\text{Bool}} \wedge \varphi_{\text{Theory}}$$

Common theories:

- **LRA:** Linear Real Arithmetic
- **BV:** Bit-Vectors (via bit-blasting)
- **UF:** Uninterpreted Functions
- **LIA:** Linear Integer Arithmetic
- **Arrays:** Read/write axioms
- **Strings:** String constraints



Major SMT solvers: Z3 (Microsoft), CVC5, Yices, MathSAT

SMT Example: Program Verification

Example: Verify: "if $x > 0$ and $y > 0$, then $x + y > 0$"

SMT formula (LIA): $(x > 0) \wedge (y > 0) \wedge \neg(x + y > 0)$

If UNSAT $\Rightarrow$ property holds!

Bit-vector theory example:

Check if `(x ^ y) & z == (x & z) ^ (y & z)` for all 32-bit integers.

Encode as: $\exists x, y, z.\, (x \oplus y) \wedge z \neq (x \wedge z) \oplus (y \wedge z)$

SMT solver (via bit-blasting): **UNSAT** $\Rightarrow$ identity holds!

Real-world usage:

- KLEE: symbolic execution for C programs
- SAGE: whitebox fuzzing at Microsoft

SMT Example: Program Verification [2]

- SeaHorn: software verification
- Dafny: verified programming language

SAT in Practice: Hardware Verification

Real-world success story:

Intel's Pentium FDIV bug (1994): $475 million recall.

Now: All major CPUs are formally verified using SAT/SMT solvers.

How it works:

1. Describe circuit behavior as Boolean formulas
2. Encode correctness properties
3. Check: "Does there exist an input that violates the property?"
4. If SAT $\Rightarrow$ bug found; if UNSAT $\Rightarrow$ property verified

Scale: Modern CPU verification involves formulas with *billions* of clauses.

SAT in Practice: Hardware Verification [2]

Techniques: bounded model checking, interpolation, abstraction refinement.

Applications Deep Dive

Bounded Model Checking (BMC)

Definition 52 (Bounded Model Checking): Verify that a property holds for *all* executions up to k steps by encoding as SAT.

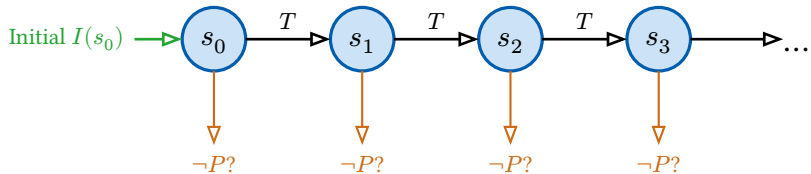
Encoding:

- State variables at each time step: $s_0, s_1, \ldots, s_k$
- Transition relation: $T(s_i, s_{i+1})$
- Initial state: $I(s_0)$
- Property violation: $\neg P(s_i)$ for some i

Formula: $I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg P(s_i)$

If **SAT** $\Rightarrow$ counterexample found!

Bounded Model Checking (BMC) [2]



Example: Verifying a mutex: "processes A and B never both in critical section."

Encode: system transitions, check if $\text{cs}_A \wedge \text{cs}_B$ is ever reachable.

Circuit SAT and Synthesis

Definition 53 (Circuit SAT): Given a Boolean circuit C, is there an input x such that $C(x) = 1$?

Note: Circuit SAT can be converted to CNF using Tseitin transformation.

Definition 54 (Circuit Synthesis): Given a specification $\varphi(x, y)$, find a circuit C such that:

$$\forall x.\, \varphi(x, C(x))$$

Approach: Use QBF or incremental SAT solving with counterexample-guided inductive synthesis (CEGIS).

CEGIS loop:

1. Propose candidate circuit C

Circuit SAT and Synthesis [2]

2. Find counterexample x where C fails
3. Add constraint ruling out this failure
4. Repeat until no counterexample exists

SAT for Cryptanalysis

SAT solvers can attack cryptographic primitives!

Example (Hash Collision): Find $x \neq y$ such that $H(x) = H(y)$.

Encode hash function as CNF, add constraint $x \neq y$ and $H(x) = H(y)$.

SAT solver searches for collision!

Applications:

- Attacking reduced-round versions of SHA, MD5
- Finding weak keys in block ciphers
- Algebraic cryptanalysis

Modern ciphers are designed to resist SAT-based attacks (high degree, many rounds).

SAT for Cryptanalysis [2]

Notable result: SAT solvers found collisions in reduced MD4 and MD5.

AI Planning with SAT

Definition 55 (Classical Planning): Given initial state I, goal G, and actions A, find a sequence of actions reaching G.

SAT encoding (SATplan):

- Variables: $\text{action}_{a,t}$ = "action a at time t", $\text{holds}_{f,t}$ = "fluent f true at t"
- Preconditions: action implies preconditions hold
- Effects: action implies effects hold at next step
- Frame axioms: unchanged fluents persist

Example: Blocks world: Stack blocks A, B, C into tower.

SAT encoding finds: pickup(A), stack(A,B), pickup(C), stack(C,A).

SATplan won the 2004 and 2006 International Planning Competitions!

SAT in Machine Learning

Decision tree learning:

Find smallest decision tree consistent with training data $\Rightarrow$ SAT/MaxSAT encoding!

Neural network verification:

Given neural network N and property P:

- Encode N as SMT formula (piecewise-linear for ReLU)
- Check: exists input violating P?

Tools: Reluplex, Marabou, α-β-CROWN

Example: Verify: "For any input image x with $\|x - x_0\| < \varepsilon$, classifier output is unchanged."

This proves *local robustness* against adversarial perturbations!

Symbolic Execution in Detail

Definition 56 (Symbolic Execution): Execute program with *symbolic* inputs, collecting *path conditions*.

Each path through the program has a constraint characterizing inputs that take it.

Example:

```
def abs(x):
    if x < 0:
        return -x
    else:
        return x
```

Path 1: $x < 0$, returns $-x$

Path 2: $x \geq 0$, returns x

Using SAT/SMT:

- Solve path conditions to generate test inputs
- Check if dangerous states (crashes, assertions) are reachable
- Find inputs triggering specific behaviors

Concolic Testing

Definition 57 (Concolic = Concrete + Symbolic): Run program with concrete inputs while maintaining symbolic constraints.

At each branch, negate a condition to explore new paths.

SAGE (Microsoft):

1. Start with random input
2. Execute symbolically, collect path constraint
3. Negate one constraint, solve with SMT
4. New input explores new path!
5. Repeat

Results:

- SAGE found 1/3 of all security bugs in Windows 7
- Runs continuously on Microsoft software
- Has found hundreds of exploitable bugs

Program Synthesis with SAT

Definition 58 (Syntax-Guided Synthesis (SyGuS)): Given a grammar G and specification φ:

Find program $P \in G$ such that $\forall x.\, \varphi(x, P(x))$

CEGIS (Counterexample-Guided Inductive Synthesis):

1. Guess candidate program P
2. Check: $\exists x.\, \neg\varphi(x, P(x))$?
3. If yes: x is counterexample, add constraint, repeat
4. If no: P is correct!

Example: Synthesize: $\max(x, y)$

Candidate 1: x — counterexample: $x = 0, y = 1$

Candidate 2: $x + y$ — counterexample: $x = 1, y = 0$

Program Synthesis with SAT [2]

Candidate 3: $\text{ite}(x \geq y, x, y)$ — correct!

SAT for Constraint Solving

Beyond Boolean constraints:

Many constraint satisfaction problems (CSPs) can be encoded as SAT!

Examples:

- **Scheduling:** Jobs, machines, time constraints
- **Resource allocation:** Assign resources without conflicts
- **Configuration:** Product configuration with compatibility rules
- **Timetabling:** University course scheduling

Encoding recipe:

1. Variables: $x_{i,v}$ = "variable i has value v"
2. At-least-one: each variable has some value
3. At-most-one: each variable has at most one value
4. Constraints: encode problem-specific rules

SAT for Constraint Solving [2]

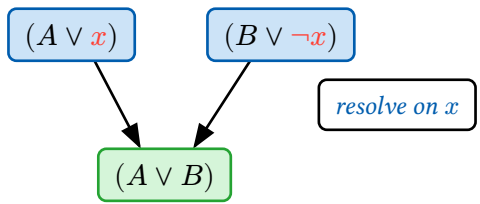
Modern CP solvers (like OR-Tools, Chuffed) often use SAT as a backend!

Theoretical Depth

Resolution Proof System

Definition 59 (Resolution rule): From $(A \vee x)$ and $(B \vee \neg x)$, derive $(A \vee B)$.

This is sound and complete for proving unsatisfiability!



Example: Prove $(x) \wedge (\neg x \vee y) \wedge (\neg y)$ is UNSAT:

1. (x) and $(\neg x \vee y) \longrightarrow (y)$ [resolve on x]
2. (y) and $(\neg y) \longrightarrow ()$ [resolve on y, empty clause!]

Empty clause $()$ means **UNSAT**.

Resolution Proof System [2]

Every CDCL run produces a resolution proof!

Learned clauses are derived via resolution from conflict clauses.

Proof Complexity

Definition 60 (Resolution width): Minimum width of any clause in a resolution proof.

Definition 61 (Resolution size): Minimum number of clauses in a resolution proof.

Theorem 8 (Width-Size Tradeoff (Ben-Sasson & Wigderson)): If resolution refutation has width w, then it has size at least $2^{\Omega(n/w)}$.

Implication: If a formula requires wide clauses to prove UNSAT, the proof must be exponentially long!

This explains why some formulas are hard for resolution-based solvers.

Example (Pigeonhole principle): PHP_n^{n+1}: $n+1$ pigeons, n holes, no two pigeons share a hole.

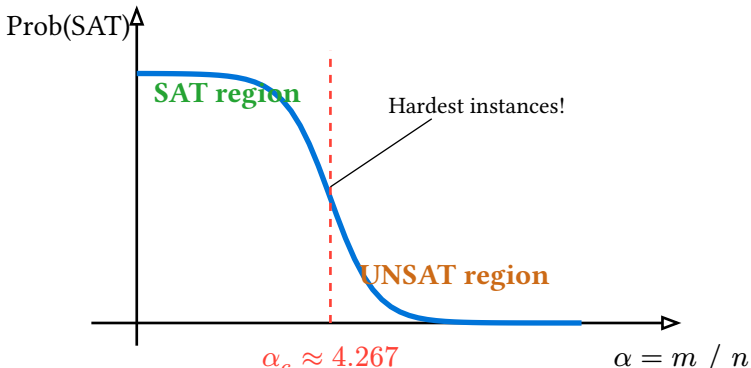
Proof Complexity [2]

Requires resolution proofs of size $2^{\Omega(n)}$!

Phase Transitions in Random SAT

Random k-SAT: Generate m random k-clauses over n variables.

Let $\alpha = m/n$ be the *clause density*.



Phase Transitions in Random SAT [2]

Theorem 9 (Phase Transition): For random 3-SAT with n variables and $m = \alpha n$ clauses:

- If $\alpha < 4.267$: almost surely **SAT**
- If $\alpha > 4.267$: almost surely **UNSAT**

The transition is *sharp* — probability jumps from near 1 to near 0!

This phase transition is analogous to physical phase transitions (like water freezing).

Lower Bounds: Why SAT is Hard

Current state of proof complexity:

No polynomial-size proofs for the pigeonhole principle in:

- Resolution (exponential lower bound)
- Bounded-depth Frege (exponential lower bound)

But we don't know if all proof systems require superpolynomial proofs!

Exponential Time Hypothesis (ETH):

3-SAT cannot be solved in time $2^{o(n)}$ where n = number of variables.

This is stronger than P $\neq$ NP and implies many other lower bounds!

Strong ETH (SETH):

Lower Bounds: Why SAT is Hard [2]

For every $\varepsilon > 0$, there exists k such that k-SAT cannot be solved in $O\left(2^{(1-\varepsilon)n}\right)$ time.

SETH implies: many “optimal” algorithms cannot be improved!

CDCL and Proof Complexity

CDCL = Resolution

Every CDCL run implicitly constructs a resolution proof!

- Decision: start a new branch
- Unit propagation: apply resolution
- Conflict analysis: derive learned clause via resolution
- Backtrack: continue proof construction

Theorem 10 (CDCL Power): CDCL with restarts can polynomially simulate *any* resolution proof.

Without restarts, CDCL is limited to *tree-like* resolution.

Implication: Lower bounds on resolution ⇒ lower bounds on CDCL!

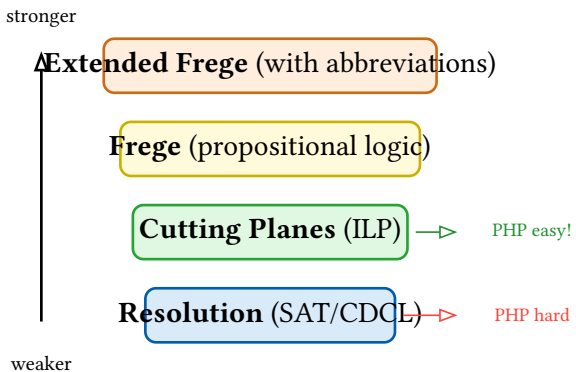
CDCL and Proof Complexity [2]

Pigeonhole principle is hard for CDCL because it's hard for resolution.

Beyond Resolution: Stronger Proof Systems

Definition 62 (Proof System Hierarchy): Resolution $<$ Cutting Planes $<$ Frege $<$ Extended Frege

Each system can simulate all weaker systems with polynomial overhead.



Beyond Resolution: Stronger Proof Systems [2]

Cutting Planes:

- Work with linear inequalities over integers
- Can prove pigeonhole with polynomial-size proofs!
- Corresponds to integer linear programming

Frege (Propositional Logic):

- Natural deduction / sequent calculus
- Can derive any tautology with short proofs
- No superpolynomial lower bounds known!

Major open problem: Are there polynomial-size Frege proofs for all tautologies?

If yes and efficiently constructible $\Rightarrow$ NP = coNP (unlikely!)

Circuit Complexity and SAT

Deep connection between SAT complexity and circuit lower bounds!

Theorem 11 (Natural Proofs Barrier (Razborov-Rudich)): "Natural" proof techniques cannot prove superpolynomial circuit lower bounds, assuming strong pseudorandom functions exist.

Why this matters for SAT:

- Proving $P \neq NP$ requires circuit lower bounds
- Natural proof barrier blocks most known techniques
- SAT algorithms might be inherently limited

Positive direction: Circuit lower bounds for ACC^0 (Carmosino et al., 2016)

Circuit Complexity and SAT [2]

Progress is slow but ongoing!

Fine-Grained Complexity

Beyond P vs NP: Study *exact* complexity of problems.

k-SAT running times:

- $k = 3$: best known $O(1.308^n)$ (Hertli, 2014)
- $k = 4$: best known $O(1.469^n)$
- General k: $O(2^{n\left(1-\frac{1}{O(k)}\right)})$

Theorem 12 (Sparsification Lemma): Any k-SAT instance with m clauses can be reduced to $2^{\varepsilon n}$ instances, each with $O(n)$ clauses, in $2^{\varepsilon n}$ time.

Consequence: Under ETH, SAT complexity depends on number of *variables*, not clauses.

Dense formulas aren't harder than sparse ones (asymptotically)!

Randomized SAT Algorithms

Definition 63 (PPSZ Algorithm (Paturi et al., 2005)): Randomized algorithm for k-SAT with best known running time for large k.

1. Randomly permute variables
2. For each variable (in order): if value is implied by unit propagation + resolution of width $\leq w$, set it; else set randomly
3. Check if assignment satisfies formula

Theorem 13 (PPSZ Running Time): For random satisfiable k-SAT instances:

Expected time $O\left(2^{n(1-\frac{\mu_k}{k})}\right)$ where $\mu_k \to \infty$ as $k \to \infty$.

For 3-SAT: $O(1.308^n)$.

Randomized SAT Algorithms [2]

Key insight: Resolution can sometimes "deduce" the correct value of a variable, reducing effective branching factor.

Schöning's Algorithm

Definition 64 (Schöning's Algorithm (1999)): Simple local search with provable guarantees:

1. Start with random assignment
2. Repeat $3n$ times:
   - If satisfying, return SAT
   - Pick any unsatisfied clause
   - Flip a random variable in it
3. Repeat whole algorithm multiple times

Theorem 14 (Schöning's Running Time): For k-SAT: expected time $O\left(\left(\frac{2(k-1)}{k}\right)^n \cdot \text{poly}(n)\right)$

For 3-SAT: $O(1.334^n)$ — worse than PPSZ but simpler!

Schöning's Algorithm [2]

Why it works: Random walk on Hamming cube. Each flip has $\geq \frac{1}{k}$ chance of moving toward solution.

Craig Interpolation

Definition 65 (Craig Interpolant): Given $A \wedge B$ is UNSAT:

An *interpolant* I is a formula such that:

- $A \to I$
- $I \wedge B$ is UNSAT
- I uses only variables common to A and B

From resolution proofs:

Given a resolution proof of $A \wedge B \Rightarrow \bot$, an interpolant can be extracted in polynomial time!

Example: $A = (x \vee y)$, $B = (\neg x) \wedge (\neg y)$

Interpolant: $I = x \vee y$ (same as A here, but uses only shared variables)

Craig Interpolation [2]

Applications:

- Model checking (unbounded verification)
- Invariant generation
- Predicate abstraction refinement

Symmetry Breaking

Definition 66 (Symmetry in SAT): Permutation of variables that maps satisfying assignments to satisfying assignments.

Problem: Symmetries cause redundant search — solver explores "equivalent" assignments.

Symmetry breaking:
1. Detect symmetries (graph automorphism)
2. Add *lex-leader* constraints: only consider "smallest" assignment in each equivalence class

Example: Graph coloring: colors are interchangeable.

Without breaking: solver tries (R,G,B), (G,R,B), (B,G,R), ...

With breaking: fix color of first vertex $\Rightarrow 6\times$ speedup!

Symmetry Breaking [2]

Tools: BreakID, Shatter, SMS (SAT Modulo Symmetries)

Modern Developments

Local Search and Incomplete Solvers

Idea: Don't prove UNSAT, just find satisfying assignments fast!

Definition 67 (WalkSAT (Selman et al., 1994)):

1. Start with random assignment
2. While unsatisfied clauses exist:
   - Pick an unsatisfied clause C
   - With probability p: flip a random variable in C
   - Otherwise: flip variable that minimizes broken clauses

Advantages:

- Very fast on satisfiable instances
- Scales to millions of variables
- Works well near phase transition

Local Search and Incomplete Solvers [2]

Disadvantages:

- Cannot prove UNSAT
- May not find solutions for hard SAT instances

Modern variants: probSAT, Sparrow, YalSAT — winners of random SAT track!

Parallel and Distributed SAT

Portfolio approach:

Run multiple solvers with different configurations in parallel. First to finish wins!

Why it works: Different solvers excel on different instances.

Divide-and-conquer:

- Split search space: $\varphi \wedge x_1 = 0$ and $\varphi \wedge x_1 = 1$
- Solve subproblems in parallel
- Challenge: balancing workload (guiding path selection)

Clause sharing:

Parallel threads share learned clauses.

Parallel and Distributed SAT [2]

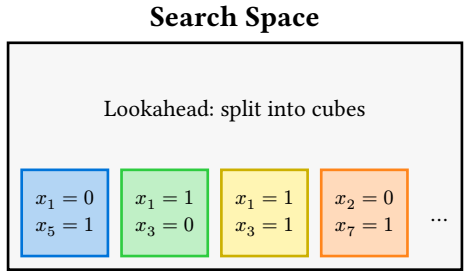
Challenge: filter useful clauses (too many $\Rightarrow$ overhead, too few $\Rightarrow$ wasted learning)

Heuristic: Share only short clauses (LBD ≤ 8).

Cube-and-Conquer

Definition 68 (Cube-and-Conquer): Hybrid parallel approach:

1. **Cube phase:** Lookahead solver partitions search space into "cubes" (partial assignments)
2. **Conquer phase:** CDCL solvers solve each cube independently



Why it works:

Cube-and-Conquer [2]

- Lookahead is good at finding hard splitting variables
- CDCL is good at solving structured subproblems
- Cubes are independent ⇒ embarrassingly parallel

Example: Pythagorean triples theorem:

- Cube phase: 800,000 cubes generated
- Conquer phase: solved on 800 cores in 2 days
- Result: UNSAT (200TB proof)

Cube-and-conquer solved several long-standing mathematical problems!

Machine Learning in SAT

Neural-guided CDCL:

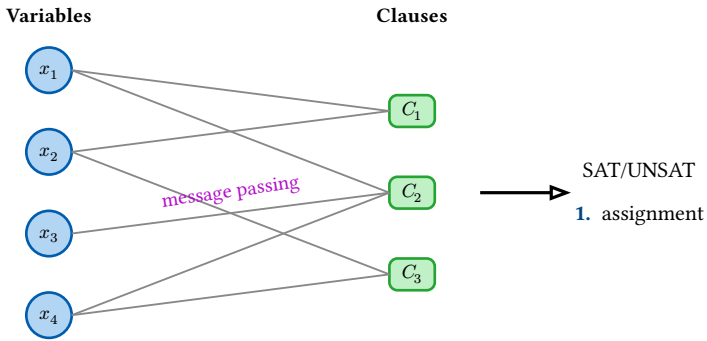
Train neural network to predict:

- Variable branching scores (replace VSIDS)
- Restart timing
- Which clauses to learn/forget

NeuroSAT (Selsam et al., 2019):

Graph neural network that predicts SAT/UNSAT directly from formula structure!

Machine Learning in SAT [2]



Current limitations:

- Still slower than engineered solvers on most benchmarks
- Requires domain-specific training
- Hard to prove correctness

Promise: Hybrid systems combining ML predictions with verified reasoning.

SAT for Combinatorics and Mathematics

SAT solvers have settled long-standing mathematical conjectures!

Notable results:

- **Boolean Pythagorean Triples (2016):** $n = 7824$ is the threshold
- **Schur Number Five (2017):** $S(5) = 160$
- **Keller's Conjecture (2020):** Disproved in dimension 8
- **Chromatic Number of the Plane:** Lower bound improved to 5 (2018)

Methodology:

1. Encode combinatorial problem as SAT
2. Use cube-and-conquer for parallelization
3. Generate machine-checkable UNSAT proof
4. Verify with certified checker

Result: Computer-assisted *proof*, not just computation!

Satisfiability Attacks

Algebraic attacks on cryptography:

Encode cipher as SAT, attack by solving!

Example: **Side-channel + SAT:**

Known: partial information about key (from power analysis)

Encode: cipher structure + partial key constraints

SAT solver: recovers full key!

SAT-based attacks succeeded against:

- Stream ciphers (Trivium, Grain)
- Reduced-round block ciphers
- Hash functions (MD4, MD5, SHA-1 reduced)

Satisfiability Attacks [2]

- Public-key primitives (factoring small RSA)

Modern ciphers are designed with SAT resistance in mind — high algebraic degree, many rounds.

SAT Solver Portfolio Selection

Definition 69 (Algorithm Selection Problem): Given instance features, predict which solver will perform best.

SATzilla approach:

1. Extract syntactic features (clause/variable ratio, graph properties, *etc.*)
2. Train classifier on solver performance data
3. At runtime: compute features, select predicted-best solver

Feature examples:

- Clause length distribution
- Variable-clause graph diameter
- Proximity to phase transition
- Horn clause fraction

SAT Solver Portfolio Selection [2]

SATzilla and successors consistently win SAT Competition portfolio tracks!

History and Key Figures

Historical Timeline

1936	Turing, Church: Undecidability of Entscheidungsproblem
1960	Davis–Putnam procedure (DP)
1962	Davis–Logemann–Loveland: DPLL algorithm
1971	Cook: SAT is NP-complete
1972	Karp: 21 NP-complete problems
1992	DIMACS format standardized
1994	Selman: WalkSAT (local search)
1996	Marques-Silva & Sakallah: GRASP (first CDCL)
1999	Moskewicz et al.: Chaff (VSIDS, 2WL)
2003	Eén & Sörensson: MiniSat
2009	Audemard & Simon: Glucose (LBD, aggressive restarts)
2016	Heule, Kullmann, Marek: Pythagorean triples (200TB proof)

Pioneers of SAT

Stephen Cook (1971)
- Proved SAT is NP-complete
- Turing Award 1982

Leonid Levin (1973)
- Independent NP-completeness proof
- “Universal search” algorithm

João Marques-Silva
- Invented CDCL (GRASP, 1996)
- Conflict clause learning
- Non-chronological backtracking

Matthew Moskewicz
- Chaff solver (1999)
- VSIDS heuristic
- Two-watched literals

Modern contributors: Armin Biere, Niklas Eén, Niklas Sörensson, Marijn Heule

The Largest Proofs Ever

Boolean Pythagorean Triples (2016):

"Can $\{1, 2, \ldots, 7824\}$ be 2-colored so no monochromatic Pythagorean triple?"

Answer: **NO**

Proof: 200 terabytes (compressed to 68GB), verified by independent checker.

Schur Number Five (2017):

"Can $\{1, 2, \ldots, 160\}$ be 5-colored with no monochromatic $a + b = c$?"

Answer: **YES** (161: NO)

Computed using massively parallel SAT solving.

The Largest Proofs Ever [2]

These are among the largest mathematical proofs ever created — generated and verified by computers!

Preprocessing and Inprocessing

Why Preprocessing Matters

Before running the main CDCL search, simplify the formula!

Goal: Remove redundant variables and clauses, making the formula smaller and easier to solve.

Preprocessing (before search):

- Variable elimination
- Subsumption
- Self-subsuming resolution
- Blocked clause elimination
- Bounded variable addition

Inprocessing (during search):

- Apply simplifications periodically
- Between restarts
- Use learned clauses
- Vivification
- Probing

Modern solvers like CaDiCaL and Kissat spend significant effort on preprocessing — it often determines success or failure!

Variable Elimination (VE)

Definition 70 (Variable Elimination): Remove variable x by resolving all clauses containing x with all clauses containing $\neg x$.

Example: Clauses with x: $(x \vee a)$, $(x \vee b)$

Clauses with $\neg x$: $(\neg x \vee c)$, $(\neg x \vee d)$

Resolvents: $(a \vee c)$, $(a \vee d)$, $(b \vee c)$, $(b \vee d)$

Remove original 4 clauses, add 4 resolvents, variable x is eliminated!

Problem: Can increase clause count! ($2 \times 2 = 4$ resolvents from 4 clauses)

Heuristic: Only eliminate x if the number of clauses decreases or stays same.

VE is the core of the SatELite preprocessor (used in MiniSat, Glucose).

Subsumption

Definition 71 (Subsumption): Clause C *subsumes* clause D if $C \subseteq D$.

If C subsumes D, then D is *redundant* and can be removed.

Example: $(x \vee y)$ subsumes $(x \vee y \vee z)$.

Remove $(x \vee y \vee z)$ — it's implied by the shorter clause!

Definition 72 (Self-Subsuming Resolution): If resolving C with D produces a clause that subsumes C, strengthen C.

Example: $C = (x \vee y \vee z)$, $D = (\neg x \vee y)$

Resolvent: $(y \vee z)$ subsumes C!

Replace C with $(y \vee z)$ (shorter clause).

Blocked Clause Elimination (BCE)

Definition 73 (Blocked Clause): Clause C with literal l is *blocked* on l if every resolvent of C on l is a tautology.

Example: $C = (x \vee a \vee b)$, and all clauses with $\neg x$ have form $(\neg x \vee \neg a \vee \ldots)$ or $(\neg x \vee \neg b \vee \ldots)$.

Every resolution produces $(a \vee b \vee \neg a \vee \ldots)$ — tautology!

C is blocked and can be safely removed.

BCE can remove many clauses without changing satisfiability!

Key insight: Blocked clauses are "useless" for unit propagation.

Bounded Variable Addition (BVA)

Definition 74 (Bounded Variable Addition): *Add* auxiliary variables to make the formula smaller!

Example: Clauses: $(a \vee b \vee c)$, $(a \vee b \vee d)$, $(a \vee b \vee e)$

Introduce $x = (a \vee b)$:

New clauses: $(x \vee c)$, $(x \vee d)$, $(x \vee e)$, $(\neg x \vee a \vee b)$

Reduced from 9 literals to 8 literals!

BVA reverses Tseitin: Instead of expanding definitions, we *create* definitions for common subexpressions.

Very effective on structured formulas from hardware verification.

Vivification

Definition 75 (Vivification): Try to shorten clauses by temporarily assuming their negation and propagating.

Algorithm:

1. For clause $(l_1 \vee l_2 \vee ... \vee l_k)$
2. Temporarily set $l_1 = 0$, propagate
3. If conflict, clause can be shortened to (l_1)
4. If l_2 propagates, continue with l_3, *etc.*
5. Result: potentially shorter clause

Example: Clause $(a \vee b \vee c)$, and $(\neg a)$ propagates to $b = 1$.

Then clause can be strengthened to $(a \vee b)$!

Vivification [2]

Vivification is expensive but very effective — used in top solvers during inprocessing.

Certified UNSAT: DRAT Proofs

Why Certify UNSAT?

Problem: SAT solvers are complex (>50,000 lines of code). How do we trust an UNSAT answer?

- Bugs in implementation
- Hardware errors (bit flips)
- Adversarial inputs

Solution: Produce a *proof* that can be independently verified!

- Solver produces proof alongside answer
- Simple checker verifies proof
- Trust shifts to small, simple checker

Since 2014, SAT Competition requires UNSAT proofs for all UNSAT answers!

Resolution Proofs

Definition 76 (Resolution Proof): A sequence of clauses where each clause is either:

- An original clause from the formula, or
- Derived by resolution from two previous clauses

A proof of UNSAT ends with the empty clause $\square$.

Example: Formula: $(x \vee y)$, $(\neg x)$, $(\neg y)$

Proof:

1. $(x \vee y)$ [original]
2. $(\neg x)$ [original]
3. (y) [resolve 1,2 on x]
4. $(\neg y)$ [original]
5. $\square$ [resolve 3,4 on y]

Empty clause $\Rightarrow$ UNSAT!

DRAT: Deletion Resolution Asymmetric Tautology

Definition 77 (DRAT Proof): A sequence of clause *additions* and *deletions*:

- Addition: Add a clause that is a Reverse Unit Propagation (RUP) or Asymmetric Tautology (RAT)
- Deletion: Remove a clause (for efficiency)

RUP (Reverse Unit Propagation):

Clause C has RUP if adding $\neg C$ (negation of all literals) leads to a conflict via unit propagation.

Intuition: C is *implied* by the current formula.

Example: Formula: $(x \vee y)$, $(\neg x \vee z)$, $(\neg y \vee z)$

Clause (z) has RUP: adding $(\neg z)$ propagates to conflict.

$(\neg z) \rightarrow$ formula becomes $(x \vee y)$, $(\neg x)$, $(\neg y) \rightarrow$ UNSAT

DRAT Checking

DRAT-trim (Heule, Hunt, Wetzler):

- Fast proof checker
- Verifies DRAT proofs in time proportional to proof size
- Used in SAT Competition since 2014

Formally verified checkers:

- **ACL2check:** Verified in ACL2 theorem prover
- **GRAT:** Verified in Isabelle/HOL
- **cake_lpr:** Verified in HOL4 with CakeML

These provide *mathematical certainty* that UNSAT answers are correct!

Trust chain: Complex solver → DRAT proof → Simple checker → Formal verification

DRAT Checking [2]

We only need to trust the *formally verified checker*, not the solver!

Proof Sizes and Compression

DRAT proofs can be huge:

- Typical: $10\times$ – $100\times$ the formula size
- Pythagorean triples: 200 TB proof!
- Compression (LRAT format): $10\times$ – $100\times$ smaller

Definition 78 (LRAT (Linear RAT)): Optimized format that includes hints for faster checking:

- Each step includes clause IDs used in derivation
- Checker doesn't need to search — just verify hints
- Enables $O(n)$ checking instead of $O(n^2)$

LRAT makes it practical to check even the largest proofs!

Model Counting and AllSAT

Beyond Satisfiability: Counting

Definition 79 (#SAT (Model Counting)): Given a CNF formula φ, count the number of satisfying assignments.

$$\#\mathrm{SAT}(\varphi) = |\{\sigma : \sigma \vDash \varphi\}|$$

Complexity: #SAT is **#P-complete** — even harder than NP-complete!

#P is the class of counting problems associated with NP decision problems.

Example: Formula: $(x \vee y) \wedge (\neg x \vee \neg y)$

Satisfying assignments: $(0,1), (1,0), (1,1)$ — wait, check $(1,1)$: $(1) \wedge (0) = 0$ ✗

Actually: $(0,1), (1,0)$ only. $\#\mathrm{SAT} = 2$.

Beyond Satisfiability: Counting [2]

Even if P = NP, #SAT would still be hard! (Unless P = #P, very unlikely)

Why Model Counting Matters

Applications:

- **Probabilistic inference:** Compute probability of events in Bayesian networks
- **Reliability analysis:** Probability that a system fails
- **Network reliability:** Probability of connectivity
- **Information leakage:** Quantify secrets revealed by program outputs
- **Quantified information flow**

Example: Bayesian network query: $P(\text{Disease} \mid \text{Symptom})$

Encode as weighted model counting:

$$P(\text{Disease} \mid \text{Symptom}) = \frac{W(\text{Disease} \wedge \text{Symptom})}{W(\text{Symptom})}$$

where $W(\varphi)$ is the weighted count of models of φ.

Model Counting Algorithms

Exact counting:

- **DPLL-based:** Exhaustive search with caching (SharpSAT, c2d)
- **Knowledge compilation:** Convert to d-DNNF, then count in linear time
- **Component caching:** Exploit formula structure

Approximate counting:

- **Hashing-based:** Add random XOR constraints, binary search
- **ApproxMC:** ε-δ approximation guarantees
- **Sampling:** Uniform sampling from solution space

Example: ApproxMC can count solutions to formulas with $10^{\{100\}}$ solutions with guaranteed accuracy $(1 \pm \varepsilon)$ and high probability!

AllSAT: Enumerating All Solutions

Definition 80 (AllSAT): Given a CNF formula φ, enumerate *all* satisfying assignments.

Blocking clause approach:

1. Find a solution σ
2. Add blocking clause $\neg\sigma$ (forbids this exact solution)
3. Repeat until UNSAT

Example: Formula: $(x \vee y)$

Solution 1: $x = 1, y = 0 \rightarrow$ add $(\neg x \vee y)$

Solution 2: $x = 0, y = 1 \rightarrow$ add $(x \vee \neg y)$

Solution 3: $x = 1, y = 1 \rightarrow$ add $(\neg x \vee \neg y)$

Now UNSAT. Total: 3 solutions.

AllSAT: Enumerating All Solutions [2]

Challenge: Number of solutions can be exponential! (up to 2^n)

Practical only for formulas with “few” solutions.

Knowledge Compilation

Definition 81 (Knowledge Compilation): Compile Boolean formula into a *tractable representation* that supports efficient queries.

Target languages:

- **BDD (Binary Decision Diagram):** DAG representation, canonical
- **OBDD:** Ordered BDD, variable order fixed
- **d-DNNF:** Decomposable Negation Normal Form
- **SDD (Sentential Decision Diagram):** Combines BDD and DNNF benefits

Tractable operations:

- Model counting: $O(|\text{compiled}|)$
- Satisfiability check: $O(1)$
- Conditioning: $O(|\text{compiled}|)$
- Model enumeration: $O(|\text{models}|)$

d-DNNF and Model Counting

Definition 82 (d-DNNF (Decomposable NNF)): Formula in NNF where for each AND node, children share no variables.

Why d-DNNF is powerful:

Model count of AND: product of children's counts!

Model count of OR: sum of children's counts!

$\Rightarrow$ Linear-time counting after compilation.

Example: $(x \wedge y) \vee (\neg x \wedge z)$ in d-DNNF:

Count = Count$(x \wedge y)$ + Count$(\neg x \wedge z) = 1 \cdot 1 + 1 \cdot 1 = 2$

(With proper handling of free variables)

d-DNNF and Model Counting [2]

Tools: c2d, D4, miniC2D compile CNF to d-DNNF for counting and queries.

More Encoding Examples

N-Queens Problem

Definition 83 (N-Queens): Place n queens on an $n \times n$ chessboard so that no two queens attack each other.

SAT encoding:

Variables: $q_{i,j}$ = "queen at row i, column j"

Constraints:

- At least one queen per row: $\bigvee_j q_{i,j}$ for each i
- At most one queen per row: $\left(\neg q_{i,j_1} \vee \neg q_{i,j_2}\right)$ for $j_1 \neq j_2$
- At most one per column: $\left(\neg q_{i_1,j} \vee \neg q_{i_2,j}\right)$ for $i_1 \neq i_2$
- At most one per diagonal: similar constraints

Example: 8-Queens: 64 variables, 1500 clauses.

SAT solver finds solution in milliseconds!

N-Queens Problem [2]

12-Queens: 14,200 solutions. AllSAT enumerates all.

Latin Square and Sudoku Variants

Definition 84 (Latin Square): An $n \times n$ grid where each row and column contains each symbol exactly once.

SAT encoding:

Variables: $x_{i,j,k}$ = "cell (i, j) contains value k"

Same constraints as Sudoku, without box constraints.

Definition 85 (Quasigroup Completion): Given a partial Latin square, complete it to a full Latin square.

NP-complete! (despite Latin squares being "easy" to construct)

Latin Square and Sudoku Variants [2]

Variants solved by SAT:

- Killer Sudoku (sum constraints)
- Futoshiki (inequality constraints)
- KenKen (arithmetic constraints)

Pigeonhole Principle: A Hard Instance

Definition 86 (Pigeonhole Principle (PHP)): Can $n+1$ pigeons be placed in n holes with at most one pigeon per hole?

Obviously UNSAT (more pigeons than holes!), but *hard for resolution*:

Any resolution proof of PHP_n^{n+1} requires $2^{\Omega(n)}$ clauses.

SAT encoding:

Variables: $p_{i,j}$ = "pigeon i in hole j"

Clauses:

- Each pigeon in some hole: $\bigvee_j p_{i,j}$
- No two pigeons share hole: $\left(\neg p_{i,j} \vee \neg p_{k,j}\right)$ for $i \neq k$

Pigeonhole Principle: A Hard Instance [2]

Example: PHP_4^5: 5 pigeons, 4 holes.

SAT encoding: 20 variables, 35 clauses.

Modern CDCL solvers: still struggle for large n!

Cryptographic Encodings

Hash function preimage:

Given output h, find input m such that $H(m) = h$.

SAT encoding:

- Variables for each bit of input m
- Encode hash computation as Boolean circuit
- Add constraints: output bits = h

Example: MD5 produces 128-bit hash.

SAT encoding of reduced MD5 (24 rounds instead of 64):

- 50,000 variables
- 200,000 clauses

SAT solvers have found collisions in reduced MD5!

Cryptographic Encodings [2]

Limitation: Full cryptographic functions are designed to resist this — SAT encoding exists but solving is infeasible.

SAT Competition and Benchmarks

The SAT Competition

Annual competition since 2002:

- Standardized benchmarks
- Categories: random, crafted, industrial/application
- Main track: solve as many instances as possible in 5000 seconds each
- Drives solver development

Track categories:

- **Main Track:** Industrial/application instances
- **Random Track:** Random 3-SAT near threshold
- **Crafted Track:** Mathematically structured instances
- **Parallel Track:** Multi-core solving
- **No-Limits Track:** Any technique allowed

Competition has driven dramatic improvements: 100× speedup since 2002!

Benchmark Sources

Industrial:
- Hardware verification (Intel, AMD)
- Software verification (CBMC, KLEE)
- Cryptographic analysis
- Planning problems
- Configuration

Crafted:
- Pigeonhole principle
- Graph coloring
- Ramsey numbers
- Pythagorean triples
- Factoring semiprimes

Key insight: Industrial instances have *structure* that solvers exploit.

Random instances at the threshold are often harder than huge industrial ones!

Evolution of Winning Solvers

Year	Winners & Innovations
2002–2004	zChaff, BerkMin (VSIDS, learning)
2005–2008	MiniSat (simplicity), PrecoSat (preprocessing)
2009–2012	Glucose (LBD, aggressive restarts)
2013–2016	Lingeling (inprocessing)
2017–2019	MapleSAT (LRB branching)
2020–2024	CaDiCaL, Kissat (Armin Biere)

Pattern: Major innovations become standard, then incremental improvements.

Recent winners: combinations of all techniques with careful engineering.

State of the Art: Kissat (2020–)

Key features of modern solvers:

- CDCL with aggressive restarts
- LBD-based clause management
- VSIDS/CHB hybrid branching
- Extensive preprocessing (VE, subsumption, BCE, vivification)
- Inprocessing during search
- DRAT proof generation

Kissat performance (SAT Competition 2023):

- Solved 250+ out of 400 industrial instances
- Average time: under 100 seconds for solved instances
- Code: 50,000 lines of C

Open challenges:

- Random SAT near threshold
- Pigeonhole variants
- Cryptographic instances
- Extremely large industrial instances ($> 10^8$ variables)

Key Takeaways

What We Learned

Core concepts:

- SAT: determining if a Boolean formula has a satisfying assignment
- CNF: standard form for SAT solvers (AND of OR clauses)
- NP-completeness: SAT is the canonical hard problem
- Cook–Levin theorem: any NP problem reduces to SAT

Algorithms:

- DPLL: complete backtracking search with propagation
- CDCL: learning from conflicts (implication graphs, 1st UIP)
- Preprocessing: variable elimination, subsumption, BCE
- Local search: WalkSAT, probSAT for satisfiable instances

Applications:

What We Learned [2]

- Hardware/software verification (bounded model checking)
- AI planning, scheduling, configuration
- Cryptanalysis and security
- Mathematical theorem proving

Extensions and Theory

Beyond SAT:

- MaxSAT: optimization over satisfiability
- QBF: quantified Boolean formulas (PSPACE-complete)
- SMT: SAT with arithmetic, arrays, strings
- #SAT: model counting (#P-complete)

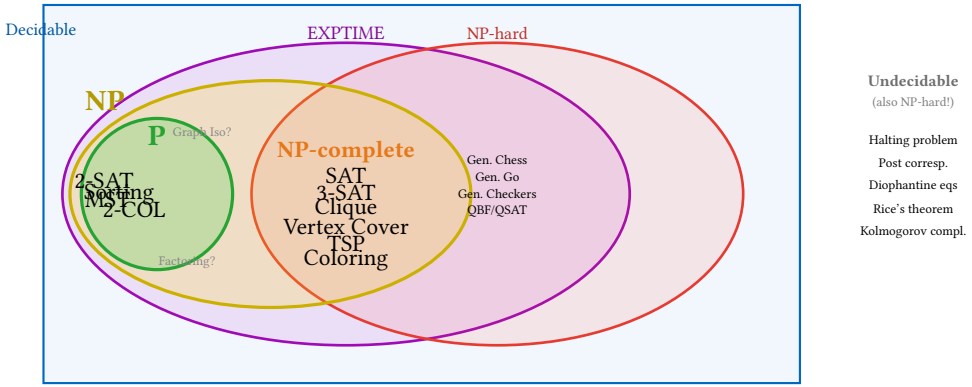
Theoretical foundations:

- Resolution proofs and proof complexity
- Phase transitions in random SAT
- ETH and SETH conjectures
- Parameterized complexity (treewidth, backdoors)

Modern developments:

- Certified UNSAT (DRAT proofs)
- Parallel SAT (cube-and-conquer)
- Machine learning in SAT (NeuroSAT)
- SAT for combinatorics (200TB Pythagorean proof!)

The Big Picture



Open question: Is P = NP? (Probably not, but unproven!)

If P $\neq$ NP, then SAT has no polynomial algorithm — but our solvers work amazingly well in practice!

The SAT Ecosystem

Major SAT solvers:

- CaDiCaL, Kissat (state of the art)
- MiniSat (educational)
- Glucose (LBD-based)
- CryptoMiniSat (XOR support)

SMT solvers:

- Z3 (Microsoft)
- CVC5
- Yices, MathSAT
- Boolector (bit-vectors)

Related tools:

- SharpSAT, D4 (model counting)
- Open-WBO, MaxHS (MaxSAT)
- DepQBF, CAQE (QBF)
- DRAT-trim, GRAT (proof checking)

Using SAT Solvers in Practice

PySAT example (Python):

```python
from pysat.solvers import Glucose3

solver = Glucose3()
# (x1 or x2) and (not x1 or x3) and (not x2 or not x3)
solver.add_clause([1, 2])
solver.add_clause([-1, 3])
solver.add_clause([-2, -3])

if solver.solve():
    print("SAT:", solver.get_model())
else:
    print("UNSAT")
```

Z3 example (Python):

Using SAT Solvers in Practice [2]

```python
from z3 import *
x, y, z = Bools('x y z')
s = Solver()
s.add(Or(x, y), Or(Not(x), z), Or(Not(y), Not(z)))
if s.check() == sat:
    print(s.model())
```

Both libraries available via `pip install python-sat z3-solver`.

Further Reading

Textbooks:

- Biere et al.: *Handbook of Satisfiability* (2nd ed., 2021)
- Arora & Barak: *Computational Complexity: A Modern Approach*
- Knuth: *The Art of Computer Programming, Vol. 4B* (Satisfiability)

Online resources:

- SAT Competition: satcompetition.org
- SAT Live!: satlive.org
- SMT-LIB: smtlib.cs.uiowa.edu

Practice:

- Try encoding Sudoku, N-Queens, graph coloring as SAT
- Use PySAT (Python), SAT4J (Java), or Z3 (Python/C++)

Further Reading [2]

- Participate in SAT Competition student tracks!