Formal Methods in Software Engineering

Boolean Satisfiability, Spring 2026

Konstantin Chukharev

SAT Encoding

Boolean Satisfiability Problem (SAT)

SAT is the problem of determining whether a given Boolean formula has a *satisfying assignment* — a mapping of truth values to variables that makes the formula true.

Definition 1 (Boolean Satisfiability (SAT)): Given a propositional formula φ over variables $X = \{x_1, \ldots, x_n\}$, decide whether

$$\exists \nu : X \to \{0, 1\}. \quad \nu \vDash \varphi$$

SAT is a *decision* problem (yes/no), but in practice we want the actual assignment — the *functional SAT* problem. SAT instances are typically given in **CNF** (conjunctive normal form): a conjunction of *clauses*, where each clause is a disjunction of *literals*.

Recall: SAT is NP-complete (Cook–Levin, 1971). Any problem in NP can be encoded as a SAT instance — making SAT solvers *universal search engines*.

The Cook–Levin Theorem: Proof Sketch

Theorem 1 (Cook–Levin (Cook 1971, Levin 1973)): SAT is NP-complete: it is in NP, and *every* problem in NP can be reduced to SAT in polynomial time.

Proof idea: Every NP problem has a polynomial-time verifier V (a Turing machine). We encode V's execution as a Boolean formula:

Variables (for T steps, S cells):

- $q_{t,s}$: machine in state s at step t
- $h_{t,p}$: head at position p at step t
- $c_{t,p,\sigma}$: cell p has symbol σ at step t

Clauses enforce valid computation:

- *Initial config* — input on tape
- *Transition function* — δ as implications
- *Acceptance* — accepting state reached

The resulting formula φ is satisfiable $\Longleftrightarrow$ there exists a certificate that V accepts. The reduction is polynomial: $O(T^2)$ clauses, where $T = p(n)$ is the verifier's runtime.

Cook–Levin makes SAT solvers *universal search engines* — any polynomially verifiable property can be checked by compiling it to SAT.

SAT Encoding Methodology

To solve a search problem with a SAT solver:

1. **Define variables** to represent the problem's *state*.
   Each variable captures a binary choice in the solution.
2. **Encode constraints** as propositional formulas.
   Express what makes a solution *valid*.
3. **Translate to CNF** (clausal form).
   Use Tseitin if needed to avoid exponential blowup.
4. **Run a SAT solver** to find a satisfying assignment or prove UNSAT.

The power of SAT: This methodology turns *any* combinatorial search problem into a standard format that state-of-the-art solvers handle efficiently.

Encoding Patterns: At-Least-One & At-Most-One

Many encoding tasks require constraining *how many* variables in a group are true.

Definition 2: *At least one* (ALO) of $x_1, \ldots, x_n$ is true:

$$(x_1 \vee x_2 \vee \ldots \vee x_n)$$

single n-literal clause

Definition 3: *At most one* (AMO) of $x_1, \ldots, x_n$ is true. *Pairwise* encoding: for each pair $i < j$, add

$$(\neg x_i \vee \neg x_j)$$

$\binom{n}{2}$ binary clauses

Note: Pairwise AMO produces $O(n^2)$ clauses. For large n, *commander–variable* or *logarithmic* encodings reduce this to $O(n)$ clauses using auxiliary variables.

Encoding Patterns: Exactly-One & Implications

Definition 4 (Exactly-One (EO)): Exactly one of $x_1, \dots, x_n$ is true: ALO $\wedge$ AMO combined.

$$\underbrace{(x_1 \vee \dots \vee x_n)}_{\text{ALO}} \wedge \underbrace{\bigwedge_{i<j} (\neg x_i \vee \neg x_j)}_{\text{AMO}}$$

Common encoding primitives:

- **Implication:** $a \to b$ becomes $(\neg a \vee b)$ — one clause.
- **If-then-else:** $\text{ite}(c, t, e)$ becomes $(\neg c \vee t) \wedge (c \vee e)$ — two clauses.
- **Mutual exclusion:** “at most one of $x_1, \dots, x_n$” — use AMO.
- **Channeling:** link two groups of variables, *e.g.*, $x_{i,j} \Longleftrightarrow y_{j,i}$.

Common pitfall: Forgetting AMO and only encoding ALO. Without AMO, the solver can set *multiple* variables true — leading to invalid solutions.

Encoding Patterns: Summary

Pattern	Clauses	Aux Vars	When to Use
ALO($x_1, \ldots, x_n$)	1	0	Something must be chosen
AMO pairwise	$\binom{n}{2}$	0	At most one choice ($n \leq 10$)
AMO commander	$O(n)$	$O(n)$	At most one choice ($n > 10$)
EO = ALO + AMO	$1 + \binom{n}{2}$	0	Exactly one choice
Implication $a \rightarrow b$	1	0	Dependency between choices
If-then-else	2	0	Conditional assignment

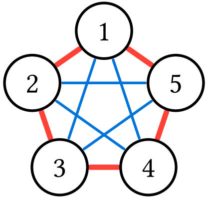
Worked Encodings

Example: Graph Coloring

Graph $G = (V, E)$: vertices V, edges E (unordered pairs). K_n — the complete graph on n vertices (every pair connected).

Problem: Color the *edges* of K_n using k colors with *no monochromatic triangle*. What is the largest n for which this is possible?

- For $k = 1$: $n = 2$ (only 1 edge).
- For $k = 2$: $n = 5$ (see diagram on the right).
- For $k = 3$: $n = 16$ — a job for a SAT solver.



Modelling Graph Coloring as SAT

1. **Variables:** For each edge e and color $c \in \{1, \ldots, k\}$, define e_c (“edge e has color c”).
2. **Constraints:**

   Each edge gets *exactly one* color (ALO + AMO):

   $$(e_1 \vee e_2 \vee e_3) \wedge \neg(e_1 \wedge e_2) \wedge \neg(e_1 \wedge e_3) \wedge \neg(e_2 \wedge e_3)$$

   No monochromatic triangle — for each triangle (e, f, g) and color c:

   $$\neg(e_c \wedge f_c \wedge g_c)$$

3. **CNF:** The constraints above are already (close to) CNF.
4. **Solve:** Increase n until the formula becomes UNSAT.

Pattern recognition: The EO constraint on edge colors is exactly the ALO + AMO pattern from the previous section.

DIMACS CNF Format

SAT solvers use the *DIMACS CNF* format — a standard text representation:

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

- `p cnf <vars> <clauses>` — header
- Variables: positive integers $1, 2, \ldots$
- Negation: prefix with `-`
- Each clause ends with `0`
- Comments start with `c`

Run a solver:

```
cadical formula.cnf
```

If SAT, the solver prints a *model* (variable assignments).
If UNSAT, it may produce a *proof certificate*.

Code: Graph Coloring SAT Encoding

```python
n = 17
k = 3
m = n * (n - 1) // 2

edges = {}
for u in range(1, n + 1):
    for v in range(u + 1, n + 1):
        edges[(u, v)] = len(edges) + 1

def color(e, c):
    return (e - 1) * k + c

clauses = []
for e in range(1, m + 1):
    # ALO: at least one color per edge
    clauses.append([
      color(e, c) for c in range(1, k + 1)
    ])
    # AMO: at most one color per edge
    for c1 in range(1, k + 1):
        for c2 in range(c1 + 1, k + 1):
            clauses.append([
```

```python
                -color(e, c1), -color(e, c2)
            ])
# No monochromatic triangles
for v1 in range(1, n + 1):
    for v2 in range(v1 + 1, n + 1):
        for v3 in range(v2 + 1, n + 1):
            e12 = edges[(v1, v2)]
            e23 = edges[(v2, v3)]
            e13 = edges[(v1, v3)]
            for c in range(1, k + 1):
                clauses.append([
                  -color(e12, c),
                  -color(e23, c),
                  -color(e13, c)
                ])
# Output DIMACS CNF
print(f"p cnf {color(m, k)} {len(clauses)}")
for clause in clauses:
    print(" ".join(map(str, clause)) + " 0")
```

Example: N-Queens (Sketch)

Place n queens on an $n \times n$ board so no two attack each other.

Variables: $q_{i,j}$ = "queen on row i, column j" (n^2 variables).

Constraints:

- **EO per row:** exactly one queen in each row i: ALO$\left(q_{i,1}, \ldots, q_{i,n}\right)$ + AMO.
- **AMO per column:** at most one queen in each column j: AMO$\left(q_{1,j}, \ldots, q_{n,j}\right)$.
- **AMO per diagonal:** at most one queen on each diagonal and anti-diagonal.

Size: n^2 variables, $O(n^3)$ clauses (pairwise AMO on each line).

Note: N-Queens is a classic SAT benchmark. For $n = 1000$, the encoding has 10^6 variables and $\sim 10^9$ clauses — but modern solvers handle it in seconds.

Example: Pigeonhole Principle (Sketch)

Place $n + 1$ pigeons into n holes, at most one pigeon per hole.

Variables: $p_{i,j}$ = "pigeon i goes into hole j" $(n(n + 1)$ variables).

Constraints:

- **ALO per pigeon:** each pigeon gets a hole: $(p_{i,1} \vee \ldots \vee p_{i,n})$.
- **AMO per hole:** each hole has at most one pigeon: $(\neg p_{i,j} \vee \neg p_{k,j})$ for $i \neq k$.

This formula is **always UNSAT** ($n + 1$ pigeons cannot fit in n holes).

Proof complexity: Resolution proofs of PHP^n_{n+1} require exponentially many steps (Haken, 1985). This is why DPLL (which implicitly produces resolution proofs) struggles with pigeonhole — and why CDCL with learned clauses does better (though it's still hard).

Encodings: Key Takeaways

The SAT encoding recipe:

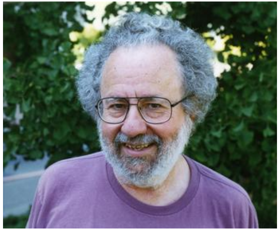
1. Identify the *choices* in your problem $\Rightarrow$ propositional variables.
2. Express *validity conditions* using ALO, AMO, EO, implication patterns.
3. Convert to CNF (usually straightforward; use Tseitin if needed).
4. Feed to a SAT solver and interpret the result.

The expressiveness of SAT encoding comes from NP-completeness: *any* polynomially verifiable problem can be encoded. The efficiency comes from modern solvers: *billions* of clauses, solved in minutes.

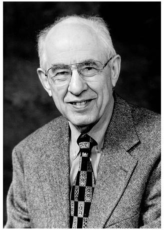
Algorithms for SAT

Davis–Putnam Algorithm

The first algorithm for SAT was proposed by Martin Davis and Hilary Putnam in 1960 [1].



Martin Davis



Hilary Putnam

Satisfiability-preserving simplification rules:

1. **Unit propagation** — propagate forced assignments.
2. **Pure literal elimination** — remove variables appearing with one polarity.
3. **Resolution** (variable elimination) — resolve away a variable.

The original DP algorithm uses resolution, which can *increase* formula size. DPLL (1962) replaces resolution with *splitting* (backtracking search), which is far more practical.

Hereinafter, formulas are given in **CNF**: a set of clauses, where each clause is a set of literals.

Unit Propagation Rule

Definition 5 (Unit clause): A *unit clause* is a clause with a single literal.

Suppose (p) is a unit clause. Recall that $\overline{p}$ denotes the complement literal: $\overline{p} = \begin{cases} \neg p \text{ if } p \text{ is positive} \\ p \text{ if } p \text{ is negative} \end{cases}$

Unit propagation:

- Assign p to true.
- Remove all clauses containing p (they are satisfied).
- Remove $\overline{p}$ from all remaining clauses (it is falsified).

Example: Consider $(A \lor B) \land (A \lor \neg B) \land (\neg A \lor B) \land (\neg A \lor \neg B) \land (A)$. The unit clause (A) forces $A = 1$. Remove clauses with A; remove $\neg A$ from the rest: $\cancel{(A \lor B)} \land \cancel{(A \lor \neg B)} \land (\cancel{\neg A} \lor B) \land (\cancel{\neg A} \lor \neg B) \land \cancel{(A)}$ Result: $(B) \land (\neg B)$ — still unsatisfiable.

Pure Literal Rule

Definition 6 (Pure literal): A literal p is *pure* if it appears in the formula only positively or only negatively.

Pure literal elimination:
- Assign the pure literal to true.
- Remove all clauses containing it (they are now satisfied).

Example: $(A \vee B) \wedge (A \vee C) \wedge (B \vee C)$. Literal A is pure (appears only positively). Assign $A = 1$, remove clauses containing A: result is $(B \vee C)$.

Note: Unit propagation is a *forced* assignment (no choice). Pure literal elimination is a *safe* assignment (any model can be extended). Both reduce the formula without branching.

Davis–Putnam–Logemann–Loveland (DPLL)

Input: set of clauses S

Output: *satisfiable* or *unsatisfiable*

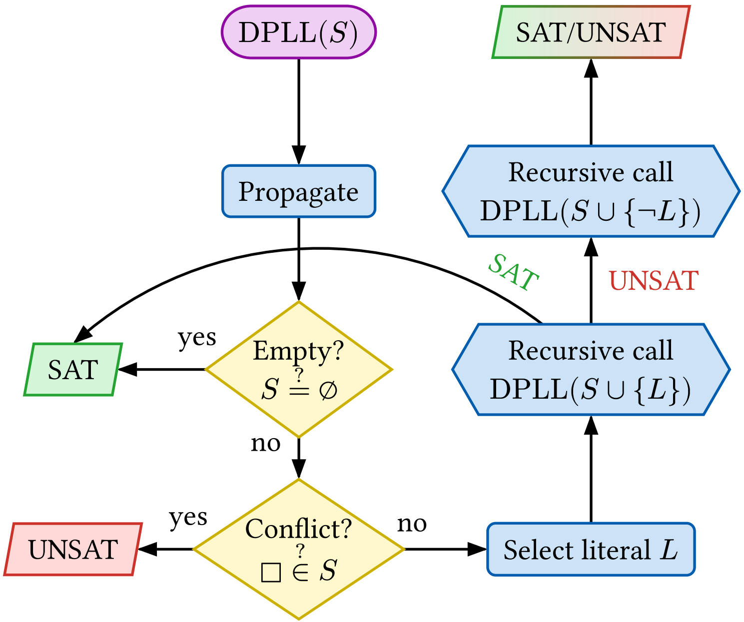
$S :=$ propagate(S)
if S is empty **then**
    return *satisfiable*
if S contains the empty clause $\square$ **then**
    return *unsatisfiable*
$L :=$ select_literal(S)
if DPLL$(S \cup \{L\}) =$ *satisfiable* **then**
    return *satisfiable*
else
    return DPLL$(S \cup \{\neg L\})$
end

DPLL [2] replaces resolution with *splitting*: choose a variable, try both values, recurse.

DPLL is *complete*: it always terminates and finds a satisfying assignment iff one exists.

The search forms a *binary decision tree* where each internal node is a variable choice and leaves are SAT/UNSAT.

DPLL Flowchart



Worked DPLL Trace

Consider: $(A \lor B) \land (\neg A \lor C) \land (\neg B \lor \neg C) \land (A \lor \neg C)$ — 4 clauses, 3 variables.

Step	Action	Assignment	Clauses	Status
1	Decide $A = 1$	$A = 1$	$(\neg A \lor C) \Rightarrow (C)$; others simplified	
2	Unit prop $C = 1$	$A = 1, C = 1$	$(\neg B \lor \neg C) \Rightarrow (\neg B)$	
3	Unit prop $B = 0$	$A = 1, C = 1, B = 0$	Check $(A \lor B)$: satisfied	SAT ✓

Model: $\nu = \{A = 1, B = 0, C = 1\}$. Verify: all clauses satisfied.

Note: In this example, DPLL found a solution without backtracking. On harder instances (*e.g.*, pigeonhole), it may explore exponentially many branches.

From DPLL to CDCL

DPLL's weakness: **chronological backtracking**.

When a conflict occurs, DPLL backtracks to the *most recent* decision — even if that decision was irrelevant to the conflict. This leads to re-exploring huge search spaces.

DPLL:

- Backtrack to the previous decision
- Undo it, try the other value
- No "memory" of *why* the conflict occurred
- May repeat the same mistake in a different subtree

CDCL insight:

- *Analyze* the conflict: which decisions caused it?
- *Learn* a new clause that prevents the same scenario
- *Backjump* to the source of the problem
- Never make the same mistake again

Key question: When a conflict occurs, can we jump back to the *cause* rather than just the last decision?

DPLL: Key Takeaways

DPLL = backtracking + unit propagation + pure literal.

- Decides a variable, propagates consequences, recurses.
- Complete: always finds a solution or proves UNSAT.
- Worst case: $O(2^n)$ — explores the full binary decision tree.
- The backbone of *all* modern SAT solvers.

Note: DPLL can be formalized as a *transition system* with rules for unit propagation, decisions, and backtracking [3]. This abstract framework extends naturally to CDCL via Learn and Backjump rules.

Conflict-Driven Clause Learning

Implication Graph

During propagation, each forced assignment has a *reason* — the clause that caused it. The **implication graph** records these dependencies.

Definition 7 (Implication Graph): A directed acyclic graph where:

- **Nodes** are assigned literals (annotated with *decision level*).
- **Edges** trace the clause that *forced* each propagated literal.
- **Decision nodes** have no incoming edges (marked with ■).
- A **conflict node** κ is added when a clause becomes empty.

Example: Suppose at decision level 3 we decide $x_1 = 1$, and this forces propagations:

- $x_1 = 1$ forces $x_4 = 1$ (via clause $c_1 : \neg x_1 \vee x_4$).
- $x_4 = 1$ and prior $x_2 = 1$ force $x_5 = 0$ (via clause $c_2 : \neg x_4 \vee \neg x_2 \vee \neg x_5$).
- $x_5 = 0$ and prior $x_3 = 1$ create a *conflict* (via clause $c_3 : x_5 \vee \neg x_3$).

The implication graph shows the chain: $x_1 \Rightarrow x_4 \Rightarrow \neg x_5 \Rightarrow \kappa$, with side edges from prior decisions x_2, x_3.

Conflict Analysis

When a conflict occurs, CDCL traces the implication graph backward to find the *root cause*.

Definition 8 (1-UIP (Unique Implication Point)): The *1-UIP* is the last decision-level node on every path from the current decision to the conflict. Cut the implication graph at the 1-UIP boundary — the literals on the *reason side* (negated) form the **learned clause**.

Example: From the previous example: the 1-UIP cut at x_4 yields the learned clause

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

This clause prevents the solver from ever simultaneously setting $x_2 = 1$ and $x_4 = 1$ again.

The learned clause is added permanently to the clause database — the solver *remembers* this conflict.

Learned clauses are resolution proofs in disguise. Each learned clause corresponds to a sequence of resolution steps on the original clauses.

Worked CDCL Example

Consider formula F with variables $x_1, \ldots, x_5$ and clauses:

$$c_1 = (x_1 \vee x_2), \quad c_2 = (\neg x_1 \vee x_3), \quad c_3 = (\neg x_2 \vee x_3), \quad c_4 = (\neg x_3 \vee x_4), \quad c_5 = (\neg x_3 \vee \neg x_4)$$

Level	Action	Propagation	Status
1	Decide $x_1 = 0$	$c_1 \Rightarrow x_2 = 1$; $c_3 \Rightarrow x_3 = 1$; $c_4 \Rightarrow x_4 = 1$; c_5 conflict: needs $\neg x_4$ but $x_4 = 1$	Conflict!
	Analyze: learned clause (x_1)		Backjump to level 0
0	Unit prop: $x_1 = 1$	$c_2 \Rightarrow x_3 = 1$; $c_4 \Rightarrow x_4 = 1$; c_5 conflict again	Conflict!
	Level 0 conflict $\Rightarrow$ **UNSAT**		

The formula is unsatisfiable. CDCL proved this by learning a clause at level 1 and detecting a conflict at level 0.

Non-Chronological Backtracking

Chronological (DPLL):

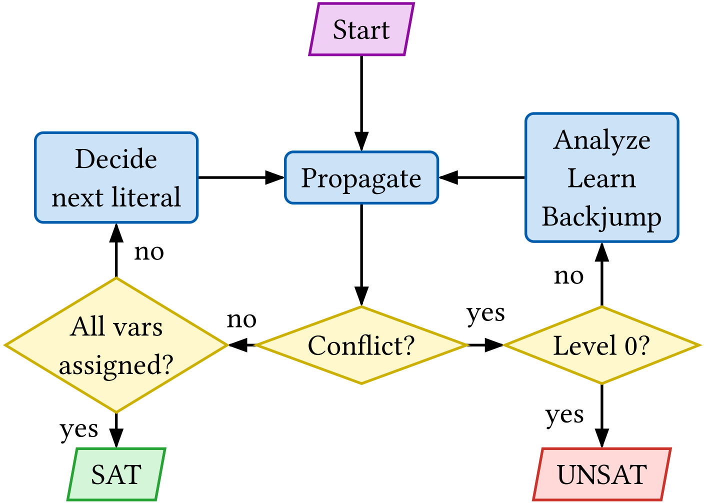
- Conflict at level k
- Go back to level $k - 1$
- Try the other branch
- May waste work if level $k - 1$ was irrelevant

Non-chronological (CDCL):

- Conflict at level k
- Analyze: learned clause has highest levels k and j ($j < k$)
- **Backjump** directly to level j
- Skip all levels between j and k

This is the key insight: Backjumping skips irrelevant search space. Combined with clause learning, CDCL avoids repeating the same mistakes. This is why CDCL solvers outperform DPLL by orders of magnitude on structured problems.

CDCL Flowchart



CDCL Heuristics

Beyond the core algorithm, several heuristics make CDCL practical:

VSIDS (Variable State Independent Decaying Sum):

- Bump the *activity score* of variables involved in recent conflicts.
- Periodically decay all scores.
- Always decide the highest-activity variable next.
- Effect: focuses search on the "hard" part of the problem.

Restarts:

- Periodically restart the search from scratch, keeping all learned clauses.
- Uses Luby sequence or geometric schedule for restart intervals.
- Effect: avoids getting stuck in unproductive search regions.

Phase saving:

- When deciding a variable, use its *last assigned polarity* as default.
- Effect: quickly reconstructs parts of previous partial solutions.

SAT Solver Architecture

A modern CDCL solver (*e.g.*, MiniSat, 2k lines of C++) consists of:

Component	Purpose
Clause database	Stores original + learned clauses; periodically garbage-collects
Two-watched-literal scheme	Efficient unit propagation — only visit a clause when a watched literal becomes false
Decision heuristic (VSIDS)	Pick the next branching variable; bump activity on conflict
Conflict analysis (1-UIP)	Derive learned clause from implication graph
Restart policy	Luby or geometric; prevents getting stuck in unproductive subtrees
Phase saving	Remember last polarity of each variable for faster re-exploration

Note: The two-watched-literal scheme is the key to scalability — it makes unit propagation amortized $O(1)$ per propagation step, instead of scanning every clause.

Modern SAT Solvers and Competitions

Key solvers:

- **MiniSat** (Eén & Sörensson, 2003): clean, educational, widely embedded.
- **CaDiCaL** (Biere): state-of-the-art, incremental, proof logging.
- **Kissat** (Biere, 2020): competition-optimized, *inprocessing* techniques.

SAT Competition (annual since 2002):

- **Industrial** track: real-world instances (verification, planning).
- **Crafted** track: hard combinatorial problems.
- **Random** track: random k-SAT near phase transition.
- Drives solver improvement; open-source requirement.

Scale: modern solvers routinely handle *millions* of variables and *billions* of clauses. From NP-complete in theory to practical workhorse — the gap is bridged by CDCL + smart engineering.

Applications of SAT

Hardware Verification

- Bounded Model Checking (BMC): unroll circuit k times, check for bugs via SAT.
- Equivalence checking: are two circuits functionally identical?
- Used at Intel, AMD, ARM for chip design validation.
- *Intel Pentium FDIV bug (1994)*: cost $475M. Modern SAT-based verification would have caught it.

AI Planning

- Encode: can we reach goal state in k steps?
- SATPlan: competitive with dedicated planners.
- Mars rover path planning: 60-minute problems solved in seconds.

Software Analysis

- Symbolic execution backends (KLEE, SAGE).
- Concolic testing: concrete + symbolic execution.
- Configuration coverage: Linux kernel has 15k+ options $\Rightarrow 2^{15000}$ configs — SAT finds bugs in specific combinations.

Cryptanalysis & Mathematics

- *Boolean Pythagorean Triples theorem (2016)*: proved using SAT solver, generated 200 TB proof!
- Attack stream ciphers via algebraic SAT encoding.
- Factorization: $n = p \times q$ encoded as multiplication circuit + SAT.

Applications of SAT [2]

In 2016, researchers used a SAT solver to solve the Boolean Pythagorean Triples problem — a 35-year-old open problem in Ramsey theory. The solver ran for 2 days on 800 cores, exploring 10^{18} search states, and produced a 200 TB proof (largest math proof ever).

- *The problem:* Can we 2-color positive integers such that no Pythagorean triple $a^2 + b^2 = c^2$ is monochromatic?
- *Answer:* Yes up to 7824, impossible beyond.

CDCL: Key Takeaways

CDCL = DPLL + clause learning + non-chronological backtracking.

- Analyzes conflicts via *implication graphs*.
- Derives *learned clauses* that prune future search.
- *Backjumps* to the relevant decision level, skipping irrelevant levels.
- VSIDS + restarts + phase saving = practical efficiency.
- Basis of *every* competitive SAT solver since 2000.

Summary and Exercises

Summary

SAT Encoding:

- Variables capture binary choices
- ALO, AMO, EO constrain cardinality
- Implication $a \to b$ = one clause
- 4-step methodology: model ⇒ constrain ⇒ CNF ⇒ solve

SAT Solving:

- DPLL: backtracking + unit propagation
- CDCL: + clause learning + backjumping
- Implication graphs trace propagation
- 1-UIP: derive learned clauses at conflicts
- VSIDS, restarts: practical performance

The pipeline: Problem ⇒ SAT encoding (ALO/AMO/EO) ⇒ DIMACS CNF ⇒ CDCL solver ⇒ model or UNSAT proof.
Next: FOL theories and SMT — extending SAT with richer background knowledge.

Exercises: SAT Encoding

1. Encode the following problem as a SAT instance: *Schedule 4 lectures into 3 time slots such that no two lectures with a shared student occur in the same slot.* Define the variables, ALO/AMO/EO constraints, and conflict constraints.
2. Write a Python script to generate the DIMACS CNF encoding for vertex coloring of a graph with n vertices, m edges, and k colors. Test it on the Petersen graph ($n = 10$, $k = 3$).
3. Show that a DNF formula can be converted to an equivalent CNF in exponential size in the worst case, but the Tseitin encoding produces an *equisatisfiable* CNF of linear size. Why does equisatisfiability (rather than equivalence) suffice for SAT solving?

Exercises: DPLL

1. Run the DPLL algorithm (with unit propagation) on the formula: $(A \lor B \lor C) \land (\neg A \lor B) \land (\neg B \lor C) \land (\neg C \lor A) \land (\neg A \lor \neg B \lor \neg C)$ Draw the search tree showing all decisions, propagations, and backtracks.
2. Explain why the pure literal rule is *sound* (preserves satisfiability) but is rarely used in modern solvers.
3. $\star$ Consider the pigeonhole formula PHP_4^3 (4 pigeons, 3 holes). How many nodes does the DPLL search tree have in the worst case? What is the optimal variable ordering?

Exercises: CDCL

1. Given the following implication graph, identify the 1-UIP and derive the learned clause:
   Decisions: $x_1 = 1$ (level 1), $x_2 = 0$ (level 2).
   Propagations: $x_3 = 1$ (from $c_1 : \neg x_1 \vee x_3$), $x_4 = 1$ (from $c_2 : x_2 \vee x_4$), conflict on $c_3 : \neg x_3 \vee \neg x_4$.

2. Explain why a conflict at decision level 0 implies UNSAT.

3. Compare DPLL and CDCL on the formula from Exercise 1 above. Does CDCL learn any useful clauses?

4. $\star$ Show that every CDCL execution on an unsatisfiable formula implicitly constructs a *resolution proof*. Explain why this means CDCL can never be worse than tree-like resolution (up to polynomial overhead).

Bibliography

[1] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, 1960, doi: 10.1145/321033.321034.

[2] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962, doi: 10.1145/368273.368557.

[3] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T)," *Journal of the ACM*, vol. 53, no. 6, pp. 937–977, 2006, doi: 10.1145/1217856.1217859.