A Deterministic Polynomial-Time Approach to SAT

Alwin
Universitas Indonesia

May 19, 2025

Abstract

The question of whether P equals NP remains one of the most profound open problems in computer science. In particular, the Boolean Satisfiability Problem (SAT) serves as the canonical NP-complete problem: if SAT admits a deterministic polynomial-time algorithm, then P = NP follows. This paper provides an English-language exposition of a proposed exploration toward proving a polynomial-time deterministic solution for SAT. The document begins with a concise overview of computational complexity theory, followed by an expanded treatment of SAT logic, including formal definitions, examples, and key logical principles. Historical approaches to SAT (DPLL, CDCL, randomized heuristics) are reviewed, along with their theoretical limitations. Potential new paradigms—ranging from hidden-structure exploitation to algebraic and topological methods—are discussed. Finally, the intellectual, technical, and psychological qualities necessary for researchers in this domain are outlined, and an exploratory roadmap for future work is proposed. The goal of this paper is both to translate and consolidate existing insights and to stimulate further research by providing detailed logical analysis of SAT and its structural properties.

1 Introduction

The P versus NP question asks whether every problem whose solution can be verified in polynomial time by a deterministic Turing machine can also be solved in polynomial time by such a machine. Formally, P is the class of decision problems solvable in deterministic polynomial time, whereas NP is the class of decision problems verifiable in deterministic polynomial time given a certificate. Since Cook and Levin independently showed that Boolean Satisfiability (SAT) is NP-complete, SAT has become the central problem in the study of P versus NP. A deterministic polynomial-time algorithm for SAT would resolve P = NP affirmatively and yield polynomial-time solutions for all problems in NP.

In this paper, we present an English-language LaTeX version of an Indonesian technical report, expanding especially the section on SAT logic. We aim to ensure that no words are truncated and that the SAT discussion includes detailed logical explanations. We begin by reviewing complexity-theoretic background, then delve deeply into the logic of SAT, followed by historical algorithmic approaches. Next, we analyze why existing approaches fail to guarantee polynomial time in the worst case and survey potential new paradigms. We conclude by outlining the qualities required for successful research in this area and propose an exploratory research roadmap.

2 Background on Complexity Theory

2.1 Classes P and NP

In computational complexity theory, the class P consists of all decision problems that can be solved by a deterministic Turing machine in time bounded by a polynomial function of the input size. Formally, a language L is in P if there exists a deterministic Turing machine M and a polynomial $p(n)$ such that for every input x, machine M halts in at most $p(|x|)$ steps and accepts x if and only if $x \in L$.

Conversely, the class NP comprises those decision problems for which a "yes" answer can be *verified* by a deterministic Turing machine in polynomial time, given an auxiliary certificate (or witness). That is, L is in NP if there exists a polynomial-time deterministic verifier $V(x, c)$ and a polynomial $q(n)$ such that

$$x \in L \quad \Longleftrightarrow \quad \exists\, c \quad (|c| \leq q(|x|) \wedge V(x, c) = \text{TRUE}).$$

Since any deterministic algorithm that solves a decision problem in polynomial time can also verify its own solutions, it follows that $\text{P} \subseteq \text{NP}$.

2.2 NP-Completeness and the Cook–Levin Theorem

A decision problem L is *NP-hard* if every problem in NP can be reduced to L via a polynomial-time many-one reduction. If, in addition, L itself belongs to NP, then L is called *NP-complete*. The Cook–Levin Theorem, independently discovered by Stephen Cook (1971) and Leonid Levin (1973), established that the Boolean Satisfiability Problem (SAT) is NP-complete. More precisely, Cook showed that for any nondeterministic polynomial-time computation, one can construct in polynomial time a Boolean formula φ that is satisfiable if and only if the computation accepts. As a corollary, if there existed a deterministic polynomial-time algorithm for SAT, then every problem in NP could be solved in deterministic polynomial time, implying $\text{P} = \text{NP}$.

3 The Boolean Satisfiability Problem (SAT)

This section provides an expanded and detailed examination of SAT logic. We begin with formal definitions of Boolean formulas, literals, clauses, and Conjunctive Normal Form (CNF). We proceed to discuss fundamental logical concepts such as truth tables, the resolution rule, and basic algorithmic operations (e.g., unit propagation). Throughout, we aim to clarify precisely what is meant by "satisfiability" and how logical structure influences algorithmic behavior.

3.1 Boolean Formulas, Literals, and Clauses

A *Boolean variable* is a variable that can take one of two values: `TRUE` or `FALSE`. A *literal* is either a Boolean variable x or its negation $\neg x$. A *clause* is a disjunction (logical OR) of one or more literals. For example, the clause

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

contains three literals: x, $\neg y$, and z. A clause evaluates to `TRUE` if and only if at least one of its literals evaluates to `TRUE` under a given assignment of Boolean variables.

A *Boolean formula* in *Conjunctive Normal Form (CNF)* is a conjunction (logical AND) of clauses. That is, a CNF formula Φ has the form

$$\Phi \ = \ C_1 \ \wedge \ C_2 \ \wedge \ \cdots \ \wedge \ C_m,$$

where each C_i is a clause. For Φ to evaluate to `TRUE`, each clause C_i must evaluate to `TRUE` under the chosen assignment. We write $\text{Vars}(\Phi)$ to denote the set of all variables appearing in Φ. If $n = |\text{Vars}(\Phi)|$, then there are 2^n possible assignments of truth values to the variables.

3.1.1 Example of a CNF Formula

Consider the CNF formula

$$\Phi = (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor x_3) \land (\neg x_1 \lor \neg x_3).$$

This formula contains three clauses:

$$C_1 = (x_1 \lor \neg x_2 \lor x_3), \quad C_2 = (\neg x_1 \lor x_2 \lor x_3), \quad C_3 = (\neg x_1 \lor \neg x_3).$$

To check whether Φ is satisfiable, one can systematically examine assignments to the variables (x_1, x_2, x_3). For instance, the assignment $x_1 = \text{TRUE}$, $x_2 = \text{FALSE}$, $x_3 = \text{TRUE}$ renders:

$$C_1 = (\text{TRUE} \lor \text{TRUE} \lor \text{TRUE}) = \text{TRUE}$$

$$C_2 = (\text{FALSE} \lor \text{FALSE} \lor \text{TRUE}) = \text{TRUE}$$

$$C_3 = (\text{FALSE} \lor \text{FALSE}) = \text{FALSE}$$

Since C_3 is `FALSE`, that assignment does not satisfy Φ. By exploring further assignments, one finds that $x_1 = \text{FALSE}$, $x_2 = \text{TRUE}$, $x_3 = \text{FALSE}$ satisfies all three clauses:

$$C_1 = (\text{FALSE} \lor \text{FALSE} \lor \text{FALSE}) = \text{FALSE},$$

which actually fails as well. Continue until discovering a satisfying assignment, such as $x_1 = \text{FALSE}$, $x_2 = \text{TRUE}$, $x_3 = \text{TRUE}$, which yields:

$$C_1 = (\text{FALSE} \lor \text{FALSE} \lor \text{TRUE}) = \text{TRUE}$$
$$C_2 = (\text{TRUE} \lor \text{TRUE} \lor \text{TRUE}) = \text{TRUE}$$
$$C_3 = (\text{TRUE} \lor \text{FALSE}) = \text{TRUE}$$

Thus Φ is indeed satisfiable.

3.2 Truth Tables and Boolean Logic

At the heart of SAT is the classical two-valued Boolean logic. A truth table for a Boolean expression enumerates all possible combinations of input values and the corresponding output of the expression. For a single variable x, its truth table is:

x	x
FALSE	FALSE
TRUE	TRUE

x	$\neg x$
FALSE	TRUE
TRUE	FALSE

For two variables x and y, the truth tables for conjunction ($\land$) and disjunction ($\lor$) are:

x	y	$x \land y$
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

x	y	$x \lor y$
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

More complex Boolean formulas can be evaluated by systematically applying these fundamental operations. In SAT, one is given a Boolean formula (often directly in CNF) and asked whether there exists an assignment of truth values to all variables that makes the formula evaluate to `TRUE`.

3.3 Formal Definition of SAT

Formally, the *Boolean Satisfiability Problem* (SAT) is defined as:

$$\text{SAT} = \{\langle \Phi \rangle : \Phi \text{ is a CNF formula that is satisfiable}\}.$$

Here, $\langle \Phi \rangle$ denotes an encoding of the CNF formula Φ as a binary string. The decision question is: *Given a CNF formula Φ (encoded as a string), is there an assignment of truth values to its variables such that $\Phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ evaluates to* `TRUE`*?*

If the answer is `YES`, we say that Φ is *satisfiable*. Otherwise, we say that Φ is *unsatisfiable*. SAT was the first problem proven to be NP-complete (Cook 1971; Levin 1973). Its NP-completeness implies that if SAT lies in P, then P = NP.

3.4 Logical Principles in SAT

We now discuss several key logical principles that underlie algorithmic techniques for SAT:

(a) **Resolution Rule.** The resolution rule is a sound and complete inference rule for propositional logic in CNF. Given two clauses

$$C_1 = (\ell \ \vee \ L), \quad C_2 = (\neg\ell \ \vee \ M),$$

where ℓ is a literal and L, M denote disjunctions of other literals, the resolution rule infers the *resolvent*

$$R = (L \ \vee \ M).$$

Intuitively, if ℓ must be true to satisfy C_1, then ℓ must be false to satisfy C_2, and vice versa; hence any satisfying assignment must satisfy at least one literal in $L \vee M$. Repeatedly applying resolution can lead to deriving the empty clause (a clause with no literals), which indicates unsatisfiability.

(b) **Unit Clause and Unit Propagation.** A *unit clause* is a clause with exactly one literal, e.g., (ℓ). If a formula contains a unit clause (ℓ), then in any satisfying assignment, literal ℓ must be assigned `TRUE` (otherwise the clause would evaluate to `FALSE`). *Unit propagation* (also called *Boolean Constraint Propagation, BCP*) systematically assigns $\ell =$ `TRUE` whenever it encounters a unit clause (ℓ), simplifies all clauses accordingly (removing those that are satisfied, and removing $\neg\ell$ from clauses where it appears), and repeats until no unit clauses remain or an empty clause is produced. Unit propagation is a fundamental subroutine in DPLL and CDCL algorithms.

(c) **Pure Literal Elimination.** A literal ℓ is *pure* in a CNF formula if ℓ appears in some clauses but its negation $\neg\ell$ does not appear in any clause. If ℓ is pure, one may safely assign $\ell =$ `TRUE` without losing any potential satisfying assignment, because setting $\ell =$ `FALSE` could only affect clauses that contain ℓ but would not help satisfy any clause containing $\neg\ell$ (since such clauses do not exist). The pure literal rule eliminates all clauses containing a pure literal.

(d) **Implication Graphs and Conflict Analysis.** Modern SAT solvers based on *Conflict-Driven Clause Learning* (CDCL) build an implication graph that captures the cascade of variable assignments under unit propagation. Each time a variable is assigned (either by decision or by propagation), an implication edge is drawn. When a conflict (empty clause) arises, the solver analyzes the implication graph to identify a subset of assignments (a conflict clause) that must be avoided in subsequent search branches. This conflict clause is then added to the formula, effectively pruning a portion of the search space.

(e) **Backtracking and Backjumping.** The search tree for SAT is formed by making decisions on variable assignments. In DPLL, if one assignment branch eventually leads to conflict (unsatisfiability), the algorithm backtracks and tries the opposite assignment for the most recently decided variable. CDCL generalizes this by *backjumping* non-chronologically

to an earlier decision level determined by conflict analysis, thereby avoiding exploring large parts of the search tree that are known to be unsatisfiable.

3.5 Formalizing SAT Logic

Let $\Phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ be a CNF formula over variables $x_1, x_2, \ldots, x_n$. Define an assignment

$$A : \{x_1, x_2, \ldots, x_n\} \longrightarrow \{\texttt{TRUE}, \texttt{FALSE}\}.$$

For a literal ℓ, which is either x_i or $\neg x_i$, we write $A(\ell)$ to mean:

$$A(\ell) \;=\; \begin{cases} \texttt{TRUE}, & \text{if } \ell = x_i \text{ and } A(x_i) = \texttt{TRUE}, \\ \texttt{TRUE}, & \text{if } \ell = \neg x_i \text{ and } A(x_i) = \texttt{FALSE}, \\ \texttt{FALSE}, & \text{otherwise.} \end{cases}$$

Each clause $C_j = (\ell_{j,1} \vee \ell_{j,2} \vee \cdots \vee \ell_{j,k_j})$ evaluates under assignment A as

$$C_j(A) \;=\; \ell_{j,1}(A) \;\vee\; \ell_{j,2}(A) \;\vee\; \cdots \;\vee\; \ell_{j,k_j}(A).$$

The entire CNF formula satisfies

$$\Phi(A) \;=\; C_1(A) \;\wedge\; C_2(A) \;\wedge\; \cdots \;\wedge\; C_m(A).$$

Hence Φ is satisfiable precisely if there exists an assignment A such that $\Phi(A) = \texttt{TRUE}$, i.e., $C_j(A) = \texttt{TRUE}$ for all $1 \le j \le m$.

3.6 Complexity of SAT and NP-Completeness

Since one can verify that an assignment A satisfies Φ by evaluating each clause in time $O(mk_{\max})$ (where $k_{\max}$ is the maximum clause length), SAT is in NP. The Cook–Levin Theorem shows that every problem in NP can be reduced to SAT in polynomial time, making SAT NP-complete. As a consequence, a deterministic polynomial-time algorithm for SAT would imply P = NP; conversely, if P $\neq$ NP, then no such algorithm can exist.

4 Historical Approaches to Solving SAT

Over the past five decades, researchers have developed a variety of algorithmic techniques to solve SAT. While these techniques are highly effective on many practical instances, none has been proven to run in polynomial time on all SAT formulas. We summarize the main historical approaches below.

4.1 Backtracking and the DPLL Algorithm

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm, introduced in the early 1960s, is a recursive backtracking procedure for CNF-SAT. The high-level pseudo-code is as follows:

Function `DPLL`(Φ):

1. **Unit Propagation.** While there exists a unit clause $\{\ell\}$ in Φ,

$$\Phi \;\leftarrow\; \text{UnitPropagate}(\ell, \Phi).$$

2. **Pure Literal Elimination.** While there exists a literal ℓ that is pure in Φ,

$$\Phi \;\leftarrow\; \text{PureLiteralAssign}(\ell, \Phi).$$

3. **Check for Empty Formula or Empty Clause.**
   (a) If Φ has no clauses (all clauses have been satisfied), **return** `TRUE`.
   (b) If Φ contains an empty clause (a clause with no literals), **return** `FALSE`.
4. **Branching.** Choose a literal ℓ from some clause in Φ (often via a heuristic). Recursively:

$$\text{return } \big(\texttt{DPLL}(\Phi \wedge \{\ell\}) \ \vee \ \texttt{DPLL}(\Phi \wedge \{\neg\ell\})\big).$$

In each recursive call, DPLL simplifies the formula by assigning $\ell = \texttt{TRUE}$ (or $\neg\ell = \texttt{TRUE}$), performing unit propagation and pure literal elimination, and then recursing. Although DPLL significantly prunes the search space via these inferences, its worst-case time complexity remains $O(2^n)$ for n variables, since in the worst case it may essentially enumerate all 2^n assignments.

4.2 Conflict-Driven Clause Learning (CDCL)

In the 1990s, researchers developed the Conflict-Driven Clause Learning (CDCL) paradigm, which extends DPLL with clause learning, non-chronological backtracking (backjumping), and sophisticated variable-decision heuristics such as VSIDS (Variable State Independent Decaying Sum). The CDCL workflow is as follows:

1. **Decision.** Assign a truth value to an unassigned variable x based on a heuristic.
2. **Propagation.** Perform unit propagation until no unit clauses remain or a conflict arises.
3. **Conflict Analysis.** If a conflict (empty clause) is found, analyze the implication graph to learn a new *conflict clause* that prevents the same conflict from reoccurring. Add this learned clause to Φ.
4. **Backjump.** Determine the highest decision level to which one must return (based on the learned clause) and backjump to that level, unassigning all variables assigned after that level.
5. **Repeat.** Continue with new decisions, propagation, and learning until the formula is either discovered to be `UNSAT` (unsatisfiable) or all variables are assigned consistently (satisfiable).

CDCL solvers, such as GRASP (Marques-Silva & Sakallah, 1996, 1999), zChaff (Zhang et al., 2001), MiniSat (2003), and many others, dominate modern SAT competitions. Empirically, CDCL can solve industrial instances with millions of clauses and hundreds of thousands of variables. Nonetheless, CDCL does not guarantee polynomial-time performance in the worst case; worst-case formulas can force the solver to generate exponentially many learned clauses or perform exponentially many branches.

4.3 Randomized and Local Search Heuristics

Randomized algorithms and local search heuristics have also been applied to SAT:

- **Schöning's Random Walk Algorithm (1999).** Starting from a uniformly random assignment for n variables, if the formula is not satisfied, pick a random unsatisfied clause and flip a random literal in that clause, repeating for a bounded number of steps. This algorithm solves k-SAT in expected time $O((2(1-1/k))^n)$; for 3-SAT, the running time is roughly $O(1.334^n)$. Although this improves over naive $O(2^n)$ enumeration for small k, it remains superpolynomial.
- **Walker-Based Heuristics (GSAT, WalkSAT).** These local search procedures start from an initial random assignment and iteratively pick a variable to flip so as to maximize the number of satisfied clauses (greedy improvement), sometimes with random steps to escape local minima. Such methods often solve large random or structured instances quickly but lack completeness: they can fail to find a satisfying assignment even if one exists, and they offer no polynomial-time guarantees.

- **Derandomized Variants.** Researchers have attempted to derandomize Schöning's algorithm by constructing deterministic "covering codes" or pseudorandom sequences of assignments. To date, the best-known deterministic algorithm for general SAT still runs in time roughly $O(1.5^n)$, far from polynomial.

4.4 Algebraic and Reductive Approaches

Some research has explored reductions from SAT to other mathematical frameworks:

- **Algebraic Encodings.** Techniques such as the Nullstellensatz method translate a CNF formula into a system of polynomial equations over a finite field. A formula is unsatisfiable if and only if the corresponding polynomial system has no common zero. While algebraic proof systems (e.g., Gröbner basis) provide proof complexity insights, they do not yield deterministic polynomial-time SAT algorithms.
- **Constraint Satisfaction Problem (CSP) Reductions.** SAT can be viewed as a special case of CSP. Conversely, some CSP instances can be reduced to SAT. Although this perspective has led to specialized SAT encodings for problems in graph theory, scheduling, and verification, no general polynomial-time algorithm has emerged.

5 Why Existing Approaches Fail to Guarantee Polynomial Time

Despite decades of algorithmic advances, no deterministic polynomial-time algorithm for SAT is known. The main barriers include:

(a) **Relativization Barriers.** Baker, Gill, and Solovay (1975) showed that there exist oracles A and B such that $\mathrm{P}^A = \mathrm{NP}^A$ but $\mathrm{P}^B \neq \mathrm{NP}^B$. This implies that any proof technique for $\mathrm{P} \neq \mathrm{NP}$ or $\mathrm{P} = \mathrm{NP}$ that "relativizes" (i.e., works in the presence of arbitrary oracles) cannot resolve the question unconditionally. Most classical algorithmic techniques up to the 1970s relativize, indicating a deep barrier.

(b) **Natural Proofs Barrier.** Razborov and Rudich (1997) formulated the concept of a *natural proof*, which abstracts many combinatorial argument techniques used to prove lower bounds on circuit complexity. They showed that any natural proof for superpolynomial lower bounds would imply the existence of strong pseudorandom generators, which in turn would break widely believed cryptographic assumptions (e.g., the hardness of factoring). Hence, natural proof techniques are unlikely to establish that SAT cannot be solved in polynomial time (if strong cryptography holds).

(c) **Worst-Case Exponential Search Space.** At its core, SAT is about exploring a space of 2^n assignments. Deterministic algorithms must somehow avoid checking an exponential fraction of assignments on certain pathological inputs. Existing methods—be they branching (DPLL/CDCL), algebraic, or local search—either explicitly or implicitly prune portions of the search space, but none can rule out exponentially many assignments in all cases. Some formulas are specifically engineered to defeat common heuristics by hiding crucial structural information (e.g., expensive unsatisfiable cores or minimal backdoors).

(d) **Limits of Heuristic Guidance.** Heuristics such as VSIDS, random branching, and variable activity scoring have dramatically improved practical performance. Nevertheless, there are explicit families of formulas (e.g., certain "Tseitin tautologies" on sparse graphs, random k-CNF at the phase transition) for which these heuristics offer no polynomial-time guarantee. In the worst case, branching decisions can force the solver to explore an exponentially large search tree.

(e) **Lack of Global Structure Awareness.** Most current SAT solvers build purely syntactic representations of the formula (e.g., clause–literal incidence lists) and rely on local inferences (unit clauses, pure literals, resolution). They lack a global, high-level perspective on the solution space (e.g., geometric shape of the solution set or high-level algebraic in-

variants). Without such global structure, it is difficult to systematically avoid exponential blowup.

6 Potential New Paradigms

Given the limitations of conventional methods, several novel and more speculative paradigms have been proposed. Although none has yielded a deterministic polynomial-time SAT solver to date, they offer possible routes for future breakthroughs.

6.1 Exploiting Hidden Structural Properties

Empirical evidence suggests that many "hard" SAT instances contain hidden structural features:

- **Backdoor Variables.** A *backdoor set* is a small subset of variables $B \subseteq \mathrm{Vars}(\Phi)$ such that once one fixes assignments to the variables in B, the remaining formula becomes "easy" (e.g., Horn, 2-SAT, or acyclic). Identifying a minimal backdoor set is itself NP-hard, but if one can efficiently discover a small backdoor for typical instances, one could solve SAT by branching only on those variables and solving the residual instance in polynomial time. Research into structural decomposition, graph partitions, and variable–clause incidence graphs aims to detect backdoors via approximation or heuristics.
- **Community Structure.** The clause–variable incidence graph of a CNF formula often exhibits a "community structure" in real-world instances, where variables and clauses are clustered into loosely connected communities. By exploiting this modular organization, one might solve each community independently or apply divide-and-conquer strategies. However, constructing guaranteed polynomial-time decompositions for all formulas remains open.
- **Unsatisfiable Cores and Minimality.** For unsatisfiable formulas, the *unsatisfiable core* is a minimal subset of clauses that is itself unsatisfiable. Some research attempts to identify large unsatisfiable cores quickly (e.g., via conflict-driven learning) to prune search. For satisfiable formulas, searching for "near-unsatisfiable" substructures may guide the solver toward a satisfying assignment by revealing tightly constrained variable clusters.

6.2 Geometric and Topological Representations

Viewing the space of all assignments $\{0,1\}^n$ as the vertices of an n-dimensional hypercube offers a geometric perspective:

- **Solution Polytopes.** Consider the set of all real-valued vectors in $[0,1]^n$ that satisfy the linear inequalities derived from each clause (e.g., a clause $(x_i \vee \neg x_j \vee x_k)$ corresponds to $x_i + (1-x_j) + x_k \geq 1$). The intersection of these half-spaces is a convex polytope whose vertices are assignments that satisfy all clauses. One might attempt to traverse this polytope or apply linear programming relaxations and rounding schemes. However, integrality gaps and the presence of "fractional" vertices typically prevent a direct polynomial-time characterization of the Boolean SAT solutions.
- **Topological Data Analysis.** By constructing a simplicial complex whose vertices correspond to assignments and where simplices connect assignments that differ in a small number of bits, one obtains a topological space encoding adjacency and connectivity of solutions. Tools from persistent homology could detect "holes" or "clusters" in the solution space that might indicate combinatorial bottlenecks. Translating these topological invariants into algorithmic shortcuts is still an open research direction.
- **Discrete Geometric Entropy.** One may define an *entropy measure* on subsets of assignments, reflecting how "spread out" satisfying assignments are in the hypercube. If one can compute or approximate this entropy efficiently, it might guide search toward

denser regions of solutions. However, computing such entropy exactly is P-hard, and approximations may not suffice to guarantee polynomial-time behavior.

6.3 Algebraic and Symbolic Methods

Algebraic encodings translate SAT into systems of equations or polynomial ideals:

- **Polynomial Nullstellensatz.** One can associate each clause to a polynomial over a finite field $\mathbb{F}$ such that a Boolean assignment corresponds to a solution in $\{0,1\}^n$. For example, the clause $(x_i \vee x_j \vee \neg x_k)$ translates to the polynomial

$$f_{i,j,k}(x_i, x_j, x_k) = (1 - (1 - x_i)(1 - x_j)x_k).$$

  The formula is unsatisfiable if and only if 1 lies in the ideal generated by all such polynomials. A Nullstellensatz proof of unsatisfiability provides polynomials g_i such that

$$\sum_i g_i(x) \cdot f_i(x) \;=\; 1,$$

  certifying that there is no common zero. Although this yields an algebraic proof system, the size of Nullstellensatz certificates can be exponentially large. Current algorithms for finding them (e.g., via Gröbner bases) take superpolynomial time in the worst case.
- **Polynomial-Equation Solvers.** Some approaches reduce SAT to solving multivariate polynomial equations over $\mathbb{F}_2$ or other fields. Linearization techniques (e.g., multilinear monomial expansions) often produce exponential blowup. Similarly, the use of semidefinite programming (SDP) relaxations (e.g., from MAX-SAT approximations) cannot guarantee an exact Boolean solution in polynomial time.
- **Abstract Algebraic Structures.** One can attempt to embed Boolean logic into more exotic algebraic objects, such as finite rings, lattices, or Boolean algebras with additional operations. The hope is that those abstract representations might reveal new invariants or transformation rules that collapse the search space. To date, no such algebraic model has yielded a general polynomial-time SAT algorithm.

6.4 Entropy-Based and Information-Theoretic Measures

The notion of *deterministic entropy* attempts to quantify the amount of information required to specify a solution:

- **Shannon versus Deterministic Entropy.** Shannon entropy measures expected information content under a probability distribution. In contrast, deterministic entropy measures, for a given formula Φ, the minimum number of bits required to pin down a satisfying assignment. If one could compute or tightly bound deterministic entropy in polynomial time, one might infer that many assignments can be ruled out simultaneously. However, computing deterministic entropy exactly is as hard as counting the number of satisfying assignments (SAT), which is P-complete.
- **Information Propagation.** Some algorithms attempt to use information from random sampling or approximate counting to guide search. For instance, belief propagation (BP) in random k-SAT attempts to estimate marginal probabilities of variables. While BP can solve certain sparse instances effectively, it fails to provide worst-case polynomial-time guarantees.

6.5 Causal Logic Networks and Graphical Models

Examining logical dependencies among variables can be framed in terms of *Bayesian networks* or *Markov Logic Networks*:

- **Bayesian Network Reductions.** One may construct a directed acyclic graph (DAG) with nodes representing variables and arcs representing implications induced by clauses. For example, the clause $(\neg x_i \vee x_j)$ implies $x_i \implies x_j$. By analyzing strongly connected components and reachability in this implication graph, one can identify forced assignments and contradictions. For Horn-SAT (where each clause has at most one positive literal), this approach yields a polynomial-time algorithm. However, arbitrary CNF formulas do not yield acyclic implication graphs, and cycles can encode arbitrary NP-hard subproblems.
- **Markov Logic and Probabilistic Inference.** Markov Logic Networks (MLNs) combine first-order logic with probabilistic graphical models. One might encode each clause as a weighted formula in an MLN and attempt to compute a most probable assignment (MAP inference). Although MAP inference in general MLNs is NP-hard, approximate inference techniques (e.g., loopy belief propagation, variational methods) can sometimes identify satisfying assignments for structured SAT instances. Yet, these methods do not guarantee polynomial-time success in the worst case.

6.6 Machine Learning–Guided SAT Solving

Recent work integrates machine learning models to guide SAT solvers:

- **NeuroSAT (Selsam et al., 2018).** NeuroSAT is a message-passing neural network trained to classify small SAT instances as satisfiable or unsatisfiable. Although NeuroSAT demonstrated the ability to predict satisfiability and even construct assignments for certain instances, its running time scales poorly with problem size and does not guarantee polynomial time.
- **Learning Heuristics.** Some solvers use reinforcement learning or graph neural networks to learn variable selection and clause deletion policies. While learned heuristics can outperform handcrafted ones on specific benchmarks, they do not alter the fundamental worst-case complexity: adversarially chosen formulas can force such heuristics into exponential-time behavior.
- **Hybrid Architectures.** By combining classical CDCL engines with neural modules that predict branching decisions, one aims to accelerate search in practice. Theoretical analysis of such hybrids remains nascent; it is not yet clear whether these methodologies can overcome exponential barriers in the worst case.

7 Deep Dive into SAT Logic and Expanded Discussion

This section delves further into the logical underpinnings of SAT. We elaborate on:

- The structure of CNF formulas and how clause interactions dictate the shape of the search space.
- Resolution proof systems and their limitations in polynomial-size proofs.
- Detailed examples of unit propagation, pure literal elimination, and conflict analysis.
- Illustrations of minimal unsatisfiable cores and their combinatorial properties.
- The role of backdoor sets and how they relate to subalgebraic structures.

7.1 Clause–Variable Interaction Graphs

Given a CNF formula Φ over variables $x_1, \ldots, x_n$, define its *bipartite incidence graph* $G_\Phi = (V_x \cup V_c, E)$ where:

$$V_x = \{v_i : 1 \leq i \leq n\}, \quad V_c = \{c_j : 1 \leq j \leq m\},$$

and there is an edge $(v_i, c_j) \in E$ if and only if variable x_i (or its negation) appears in clause C_j. An alternative representation is the *primal graph* (or variable–interaction graph), whose vertices are variables x_i and edges connect x_i and x_j if they appear together in some clause.

- If the primal graph has bounded treewidth, one can solve SAT in polynomial time via dynamic programming over tree decompositions. In particular, if the primal graph is a tree (treewidth 1), SAT reduces to Horn-SAT or 2-SAT, both solvable in polynomial time. However, computing treewidth is NP-hard in general, and many SAT instances have large treewidth.
- The incidence graph can also exhibit modular structure. For example, in industrial verification instances, the graph often decomposes into loosely connected subgraphs. One can attempt to identify community partitions using spectral clustering or modularity optimization, then solve each component in isolation followed by global consistency checks. Although promising in practice, worst-case instances can force these methods to combine exponentially many components.

7.2 Resolution Proof Complexity

The resolution proof system is central in proof complexity, providing both lower bounds for unsatisfiable formulas and insights into solver behavior.

- A *resolution proof* for an unsatisfiable CNF formula Φ is a sequence of clauses $D_1, D_2, \ldots, D_t$ such that for each D_k, either D_k is an original clause of Φ or D_k is derived by resolving two earlier clauses. The goal is to derive the empty clause. The *size* of a resolution proof is the number of resolution steps (or number of clauses) in the proof.
- Haken (1985) proved that certain families of formulas (the pigeonhole principle formulas) require exponential-size resolution proofs. More generally, Urquhart (1995) and others established exponential lower bounds on tree-like and general resolution proofs for various formulas. Since CDCL solvers implicitly construct resolution proofs (learning conflict clauses corresponds to deriving resolvents), these lower bounds imply that certain families of unsatisfiable formulas force CDCL to generate exponentially many learned clauses.
- Conversely, for formulas with "small" resolution refutations, CDCL solvers can find a proof in time polynomial in the size of that proof, though not necessarily guaranteeing a priori that one exists for all formulas. This shows a deep connection between proof complexity and solver complexity: polynomial-size proofs imply polynomial-time solving in the worst case for that formula family.

7.3 Detailed Example: Unit Propagation and Conflict

Consider the following CNF formula:

$$\Phi = (x \vee y) \ \wedge \ (\neg x \vee z) \ \wedge \ (\neg y \vee z) \ \wedge \ (\neg z).$$

We walk through a DPLL-style exploration with unit propagation:

1. Initially, Φ has the unit clause $(\neg z)$. Hence, by unit propagation, assign $z = \texttt{FALSE}$ and simplify:
$$(\neg x \vee z) \ \mapsto \ (\neg x \vee \texttt{FALSE}) = (\neg x), \quad (\neg y \vee z) \ \mapsto \ (\neg y).$$
The other clause $(x \vee y)$ remains unchanged. So the simplified formula becomes
$$\Phi' \ = \ (x \vee y) \ \wedge \ (\neg x) \ \wedge \ (\neg y).$$
Now we have two unit clauses $(\neg x)$ and $(\neg y)$. We propagate $(\neg x)$, assigning $x = \texttt{FALSE}$, which simplifies $(x \vee y)$ to $(\texttt{FALSE} \vee y) = (y)$. Then we propagate $(\neg y)$, assigning $y = \texttt{FALSE}$, which simplifies (y) to $(\texttt{FALSE})$, an empty clause emerges, indicating *conflict*.
2. At this point, conflict analysis would identify that the assignments $z = \texttt{FALSE}$, $x = \texttt{FALSE}$, and $y = \texttt{FALSE}$ together cause the empty clause. A conflict clause can be learned by resolving the original clauses that participated in the unit propagations:
$$(\neg z) \quad , \quad (\neg x \vee z) \quad , \quad (\neg x),$$

which yields the learned clause $(\neg z \vee x)$ (intuitively, "if $z =$ `FALSE` then x must be `TRUE`]).*Similarly, resolving*$(\neg z)$ with $(\neg y \vee z)$ and $(\neg y)$ yields $(\neg z \vee y)$. Since x and y both appear in $(x \vee y)$, further resolution can derive $(\neg z \vee x \vee y)$. The precise conflict clause depends on the chosen learning scheme (e.g., 1-UIP).

3. With a learned clause in hand, a CDCL solver can backjump to the appropriate decision level. If $z =$ `FALSE` was forced by a decision or was a propagated assignment at decision level 0, then the solver would conclude Φ is `UNSAT`. In a DPLL (non-CDCL) solver, one would simply backtrack on the last decision; here, since there was no explicit decision before unit propagation, one concludes unsatisfiability directly.

7.4 Minimal Unsatisfiable Cores and Their Structure

A formula Φ is *minimally unsatisfiable* if Φ is unsatisfiable, but removing any clause from Φ makes it satisfiable. Studying minimally unsatisfiable formulas provides insight into the combinatorial hardness of SAT:

- For a minimally unsatisfiable formula with m clauses over n variables, a classic result (Tarsi's Lemma) states that $m \geq n+1$. Intuitively, there are more constraints than degrees of freedom. Constructing explicit families of minimally unsatisfiable formulas (e.g., hitting formulas or Tseitin tautologies) yields hard instances for resolution and CDCL.
- Identifying an unsatisfiable core (not necessarily minimal) can be done by tracking which original clauses contributed to the derivation of the empty clause during conflict analysis. While computing a minimal unsatisfiable core is NP-hard, obtaining an approximate core suffices to prune search in many solvers.
- The structure of unsatisfiable cores can highlight "bottleneck" subformulas. For instance, a large cactus-like core (where clauses overlap primarily on small subsets of variables) may indicate that one needs to flip assignments in a carefully coordinated manner—beyond simple local repairs—to escape unsatisfiability.

7.5 Backdoor Sets and Subformula Reduction

A set of variables B is a *backdoor set* for Φ relative to a polynomial-time decidable class $\mathcal{C}$ (e.g., 2-SAT, Horn, XOR-SAT) if, once one assigns values to all variables in B, the simplified formula $\Phi|_B$ belongs to $\mathcal{C}$ and hence is solvable in polynomial time. Formally:

$$\Phi|_B = \{\, C \setminus \{\ell : \ell \text{ is falsified by the assignment to } B\} \,\}$$

after removing all clauses satisfied by the assignment to variables in B. If $|B| = k$, then one can solve Φ by enumerating all 2^k assignments to B and, for each assignment, solve the resulting $\mathcal{C}$-instance in polynomial time. Thus if B is small (e.g., $k = O(\log n)$), this yields a polynomial-time algorithm. However:

- Finding a smallest backdoor set is itself NP-hard. Researchers have developed parameterized algorithms that find backdoor sets of size k in time $O^*(c^k)$, but this is exponential in k. If k grows linearly in n (as in worst-case formulas), no polynomial-time result follows.
- Some classes of formulas admit small backdoors naturally (e.g., instances from hardware verification often reduce to Horn after setting a few key variables). Quantifying the backdoor size for random or adversarial formulas remains an active area of investigation.

8 Required Qualities for SAT and P vs NP Research

Researchers aiming to prove that SAT is solvable in deterministic polynomial time (i.e., that P = NP) must combine several intellectual, technical, and psychological attributes:

(a) **Deep Theoretical Understanding.** One must be fluent in:

- Formal models of computation (Turing machines, Boolean circuits, RAM models).
- Complexity classes, reductions, and proof complexity (resolution, cutting planes, communication complexity).
- Algebraic complexity (polynomial representations, Gröbner bases) and proof systems (Nullstellensatz, polynomial calculus).
- Graph theory, spectral methods, and structural parameters (treewidth, clique-width).
- Information theory (Shannon entropy, Kullback–Leibler divergence, deterministic entropy).
- Algebraic geometry and combinatorial topology (simplicial complexes, homology).

(b) **Technical Proficiency.** A researcher must be skilled in:
- Algorithm design (branching algorithms, parameterized complexity, approximation algorithms).
- Implementation of SAT solvers (DPLL, CDCL engines) and empirical benchmarking.
- Building and analyzing data structures for graph algorithms, incidence graphs, and clause databases.
- Experimental platforms for large-scale SAT instances, including parallel and distributed solvers.
- Machine learning frameworks for learning heuristics or graph embeddings (e.g., PyTorch, TensorFlow, graph neural networks).

(c) **Collaborative and Interdisciplinary Mindset.** P vs NP research often crosses traditional boundaries. Collaborations between:
- Theoretical computer scientists and mathematicians (proof complexity, combinatorics).
- Algebraists (commutative algebra, algebraic geometry) and logicians (model theory, proof systems).
- Statisticians and data scientists (entropy measures, sampling, approximate counting).
- Machine learning researchers (neural heuristics, reinforcement learning).
- Practitioners from hardware verification, formal methods, and constraint programming.

(d) **Persistence and Psychological Resilience.** The P vs NP problem has resisted solutions for decades. Researchers must:
- Accept the possibility of repeated setbacks and negative results.
- Maintain patience and humility, recognizing that incremental progress (e.g., lower bounds for restricted models) can still yield valuable insights.
- Cultivate creativity to envision nontraditional approaches (e.g., geometric topology, causal inference).
- Retain optimism without falling to overconfidence, given the rich history of proposed "proofs" that turned out to be incorrect.

(e) **Imagination and Creativity.** Breaking the P vs NP barrier likely requires insights outside conventional complexity-theoretic frameworks. Creative leaps might involve:
- Inventing new algebraic or combinatorial invariants that tightly characterize SAT complexity.
- Developing novel topological descriptors of solution spaces and mapping them to tractable problems.
- Conceiving hybrid symbolic–numeric methods that harness continuous relaxations while preserving discrete correctness.
- Pioneering information-theoretic frameworks that quantify logical constraints in an actionable manner.

9 Exploratory Research Roadmap

Based on the survey of existing methods and proposed paradigms, we outline a multi-phase roadmap for exploratory research aimed at discovering a deterministic polynomial-time algorithm for SAT (or, alternatively, establishing stronger evidence that none exists).

9.1 Phase I: Comprehensive Literature Reconciliation

1. **Survey Foundational Texts.** Thoroughly review classical references:
   - Sipser, M. (2012). *Introduction to the Theory of Computation.* Cengage Learning.
   - Arora, S., & Barak, B. (2009). *Computational Complexity: A Modern Approach.* Cambridge University Press.
   - Papadimitriou, C. (1994). *Computational Complexity.* Addison-Wesley.
   - Cook, S. A. (1971). "The complexity of theorem-proving procedures." In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 151–158.
   - Levin, L. A. (1973). "Universal sequential search problems." *Problems of Information Transmission*, 9(3):265–266.
2. **Review Modern Surveys.** Examine contemporary research on proof complexity, parameterized complexity, and SAT-solver heuristics:
   - Razborov, A. A., & Rudich, S. (1997). "Natural proofs." *Journal of Computer and System Sciences*, 55(1):24–35.
   - Baker, T., Gill, J., & Solovay, R. (1975). "Relativizations of the P =? NP question." *SIAM Journal on Computing*, 4(4):431–442.
   - Marques-Silva, J. P. M., & Sakallah, K. A. (1996). "GRASP: A search algorithm for propositional logic." In *Proceedings of ICCAD*, 28–31.
   - Selsam, D., Bjørner, N., Liang, P., & Song, D. (2018). "Learning a SAT solver from single-bit supervision." In *International Conference on Machine Learning (ICML).*
3. **Consolidate Key Insights.** Summarize known lower bounds in proof complexity (e.g., pigeonhole principle, clique tautologies), parameterized complexity results (e.g., W[1]-hardness of certain SAT variants), and structural theorems (e.g., Schaefer's dichotomy theorem for Boolean CSP).

9.2 Phase II: Structural and Topological Analysis of SAT Instances

1. **Data Collection and Benchmarking.** Assemble a diverse SAT benchmark suite including:
   - Industrial CNF instances from hardware and software verification (e.g., from the SAT Competition archives).
   - Random k-SAT formulas at phase transition densities.
   - Crafted hard instances (e.g., Tseitin formulas on expander graphs, cryptographic combiners).
2. **Graph-Theoretic Characterization.** For each instance, compute structural graph parameters:
   - Treewidth and pathwidth of the primal graph (exact or approximate).
   - Community structure metrics (e.g., modularity, conductance).
   - Variable–clause incidence density, maximum degree, and spectrum of adjacency matrices.
3. **Topological Data Analysis.** Construct simplicial complexes of solution subspaces:
   - Build a graph H where each vertex is a candidate assignment (e.g., among samples), with edges connecting assignments at Hamming distance 1 that both satisfy Φ.
   - Compute persistent homology (Betti numbers) at varying Hamming-distance thresholds to detect clusters, holes, or bottlenecks in the solution space.

- Correlate topological features with solver performance (e.g., number of conflicts, learned clause sizes).

4. **Entropy Estimation.** Approximate (via sampling or Monte Carlo) the logarithm of the number of satisfying assignments (log of SAT) to gauge the "size" of the solution space. Evaluate whether high-entropy instances correlate with greater difficulty in practice.

9.3 Phase III: Algebraic and Logic-Based Prototyping

1. **Explore Algebraic Representations.** Develop prototypes that map SAT instances to:
   - Polynomial systems over finite fields, using Nullstellensatz certificates and polynomial calculus.
   - Linear or semidefinite relaxations and subsequently apply rounding or cutting-plane methods.
   - Gröbner basis computations with heuristics to detect early refutations.
2. **Implement Enhanced Resolution Variants.** Experiment with new resolution-based proof systems:
   - Weakening the clause learning strategies (e.g., guided by algebraic or topological invariants).
   - Augmenting resolution with additional inference rules derived from structure (e.g., generalized clause absorption, blocked clauses elimination).
   - Investigate iterative refinement: generate a sequence of formulas $\Phi_0, \Phi_1, \ldots$ where each Φ_{i+1} includes new constraints gleaned from partial algebraic or topological analysis of Φ_i.
3. **Search for Deterministic Entropy Bounds.** Design algorithms that, given Φ, estimate a deterministic lower bound on the number of assignments one must examine. If this bound can be shown to be polynomially bounded in special formula families, it could yield new algorithmic insights.

9.4 Phase IV: Machine Learning–Guided Heuristics and Hybrid Solvers

1. **Train Graph Neural Networks.** Use graph neural networks (GNNs) on the clause–variable incidence graph to predict:
   - Which variables are likely to appear in small backdoor sets.
   - Which clauses are likely to belong to an unsatisfiable core.
   - Variable assignments that maximize propagation effects.
2. **Reinforcement Learning for Branching Decisions.** Frame SAT solving as a sequential decision process where an agent chooses variables to assign at each step. Reward signals can be derived from reductions in formula size or conflict frequency.
3. **Hybrid Architectures.** Build prototypes that integrate:
   - A classical CDCL or DPLL engine.
   - Neural modules that predict promising variable assignments or clause deletions.
   - Algebraic subroutines that periodically analyze the formula's ideal structure to generate additional constraints.
4. **Evaluate on Benchmarks.** Compare performance against state-of-the-art solvers on standard benchmarks. Analyze whether the hybrid approach reduces average-case complexity, pushes the boundary of tractable instances, or hints at structural conditions that permit polynomial-time solving.

9.5 Phase V: Theoretical Consolidation and Complexity Bounds

1. **Parameter-Tailored Complexity.** For specific structural parameters (e.g., treewidth t, backdoor size k), attempt to prove fixed-parameter tractability (FPT) results of the

form $O(f(t) \cdot n^{O(1)})$ or $O(g(k) \cdot n^{O(1)})$. Seek to extend these to larger parameter classes or identify thresholds beyond which polynomial-time collapses occur.

2. **Lower Bound Exploration.** Attempt to strengthen known proof-complexity lower bounds by constructing families of formulas resistant to new inference rules. Show that no polynomial-size proofs exist in extended proof systems under plausible complexity assumptions.
3. **Conditional Impossibility Results.** If no polynomial-time algorithm emerges, clarify the barriers by proving that certain widely believed cryptographic or complexity-theoretic assumptions (e.g., Exponential Time Hypothesis, random k-SAT hardness) imply that SAT cannot be solved in polynomial time. Frame these results to highlight why new paradigms—beyond relativizing or natural proofs—are required.

9.6 Phase VI: Collaborative Platform and Community Engagement

1. **Open-Source Repository.** Establish a public repository to host:
   - Benchmark instances, along with structural and topological metadata.
   - Code for algebraic, topological, and machine-learning–guided prototype solvers.
   - Scripts to reproduce experiments, graphs, and results.
   - Documentation of negative results to inform the community of unfruitful directions.
2. **Workshops and Challenges.** Organize workshops that bring together:
   - Experts in proof complexity to discuss new lower bounds.
   - Mathematicians specializing in algebraic geometry and topology to propose novel SAT encodings.
   - Machine learning researchers to share insights on graph-based neural models.
   - Practitioners in formal methods to supply real-world benchmarks and define performance metrics.
3. **Incentivized Competitions.** Create targeted challenges, such as:
   - "Find a polynomial-time algorithm for SAT restricted to formulas with bounded topological genus of the incidence graph."
   - "Identify a structural parameter that guarantees polynomial-time solvability for CNF formulas beyond known tractable classes."
   - "Demonstrate a new proof system that yields subexponential-size refutations for pigeonhole principle formulas."

10 Conclusion

The question of whether P = NP, and in particular whether SAT admits a deterministic polynomial-time algorithm, remains open and challenging. In this paper, we have translated and expanded upon an Indonesian technical report into an English LaTeX document, paying special attention to fully elaborating on SAT logic. We reviewed the definitions of Boolean formulas, CNF, clauses, literals, resolution, and fundamental inference rules such as unit propagation and pure literal elimination. We surveyed historical approaches—including DPLL, CDCL, randomized heuristics, and algebraic methods—and explained why these methods fail to ensure polynomial-time performance in the worst case.

We then proposed potential new paradigms, ranging from exploiting hidden structural properties to adopting geometric, topological, and algebraic frameworks. We outlined a comprehensive roadmap for future research, emphasizing interdisciplinary collaboration, theoretical consolidation, and empirical benchmarking. Ultimately, solving SAT in deterministic polynomial time—if possible—will require breakthroughs that transcend current barriers such as relativization and natural proofs. Alternatively, establishing stronger impossibility results (conditional on widely believed assumptions) could provide a clearer understanding of the inherent difficulty of SAT.

We hope this document, with its expanded SAT logic discussion and detailed research plan, serves as both a synthesis of known results and a stimulus for innovative approaches. The P versus NP question stands at the heart of theoretical computer science; its resolution, in either direction, would reshape our understanding of computation, optimization, and the limits of algorithmic power.

References

1. Baker, T., Gill, J., & Solovay, R. (1975). "Relativizations of the P = ? NP question." *SIAM Journal on Computing*, 4(4), 431–442.
2. Cook, S. A. (1971). "The complexity of theorem-proving procedures." In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (pp. 151–158).
3. Haken, A. (1985). "The intractability of resolution." *Theoretical Computer Science*, 39, 297–308.
4. Levin, L. A. (1973). "Universal sequential search problems." *Problems of Information Transmission*, 9(3), 265–266.
5. Marques-Silva, J. P. M., & Sakallah, K. A. (1996). "GRASP: A search algorithm for propositional logic." In *Proceedings of ICCAD* (pp. 28–31).
6. Papadimitriou, C. H. (1994). *Computational Complexity.* Addison-Wesley.
7. Razborov, A. A., & Rudich, S. (1997). "Natural proofs." *Journal of Computer and System Sciences*, 55(1), 24–35.
8. Selsam, D., Bjørner, N., Liang, P., & Song, D. (2018). "Learning a SAT solver from single-bit supervision." In *Proceedings of the 35th International Conference on Machine Learning (ICML).*
9. Sipser, M. (2012). *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning.
10. Urquhart, A. (1995). "The complexity of propositional proofs." *Bulletin of Symbolic Logic*, 1(4), 425–467.
11. Zhang, L., Madigan, C. F., Moskewicz, M. W., Malik, S. (2001). "Efficient conflict driven learning in a Boolean satisfiability solver." In *Proceedings of ICCAD* (pp. 279–285).