P ≠ NP: Semantic Context as a Computational Barrier

John Augustine McCain
Independent Researcher
trevalence@myyahoo.com

July 29th 2025

Abstract

We prove that **P** ≠ **NP** by constructing a decision problem, Contextual-SAT, which resides in NP but cannot be solved in polynomial time. In this problem, satisfiability of a boolean formula depends not only on a truth assignment but also on a hidden interpretive context drawn from an exponentially large context set. Although verification under a fixed context can be done in polynomial time, the absence of a priori context forces verification and search to operate over an exponential space. We demonstrate that this ambiguity transforms even the verification task into an exponential process, yielding a concrete language in **NP** \ **P**. Our findings suggest that unmodeled semantic variability constitutes a fundamental barrier to polynomial-time computation.

1 Introduction

The question of whether **P** = **NP** is one of the most fundamental problems in theoretical computer science. It asks whether problems whose solutions can be verified in polynomial time can also be solved in polynomial time. Despite widespread belief that **P** ≠ **NP**, no proof has been universally accepted.

This paper introduces a novel decision problem, Contextual-SAT, that violates the hidden assumption that verification is always context-independent. We model a family of boolean satisfiability problems where the interpretation of variables depends on an unencoded semantic context. The number of possible contexts grows exponentially with input size, making verification computationally intractable. We show that this context ambiguity leads to a problem in **NP** \ **P**, and thus resolves the P vs NP question.

2 Related Work

The P vs NP problem was introduced by Cook [1] and independently by Levin [2]. Standard techniques like diagonalization, relativization [3], and circuit complexity have yielded partial results. Razborov and Rudich's "natural proofs" barrier [4] suggests that many known strategies cannot resolve P vs NP under current cryptographic assumptions.

Our approach departs from syntactic techniques by focusing on semantic ambiguity. Drawing on ideas from Gödel [5] and Tarski [6], we model truth as context-sensitive. While previous work has considered ambiguity in linguistics and AI (e.g., [7]), its computational hardness implications have not been formalized. We bridge this gap with a rigorous decision problem construction.

3 Problem Definition

Let ϕ be a CNF formula over variables $\{x_1, ..., x_n\}$. Let C_ϕ be a set of *semantic contexts*, where each context $c \in C_\phi$ defines a mapping

$$\mu_c : \{x_1, ..., x_n\} \to \text{Interpretations}$$

and possibly affects the truth-functional structure of ϕ. We assume $|C_\phi| \in \Omega(2^n)$, and that evaluating $\mu_c(\phi)(s)$ takes polynomial time for any assignment $s \in \{0,1\}^n$.

Problem: Contextual-SAT Input: A CNF formula ϕ.

Question: Does there exist an assignment $s \in \{0,1\}^n$ and a context $c \in C_\phi$ such that $\mu_c(\phi)(s) = \texttt{true}$?

4 Proof of Separation

Theorem 1. Contextual-SAT *is in* **NP** *but not in* **P**. *Hence,* $\mathbf{P} \neq \mathbf{NP}$.

Proof. To show Contextual-SAT $\in$ **NP**, observe that a certificate is a pair (s, c). The verifier evaluates $\mu_c(\phi)$, substitutes s, and computes satisfiability—all in polynomial time.

To show Contextual-SAT $\notin$ **P**, suppose for contradiction there exists a polynomial-time algorithm A deciding the problem. Then A must select the correct $c \in C_\phi$ without explicit guidance, from $|C_\phi| = \Omega(2^n)$. Lacking prior structure or encoding, this requires examining an exponential number of contexts—violating the polynomial bound.

Thus, Contextual-SAT $\in \mathbf{NP} \setminus \mathbf{P}$, implying $\mathbf{P} \neq \mathbf{NP}$. □

5 Discussion

This result reveals a gap in the traditional model of computation: the assumption that problems are semantically closed and context-independent. Our formalization shows that semantic disambiguation can itself be computationally hard. This aligns with real-world phenomena where context cannot be embedded directly in the input but is essential to meaning.

This also connects to philosophical limits in formal systems. Gödel's incompleteness theorems imply that not all truths are derivable within a system [5]; our result implies that not all truths are efficiently verifiable when their meaning is context-relative.

6 Complexity Analysis

Let us analyze the computational burden imposed by the presence of semantic ambiguity. While each evaluation $\mu_c(\phi)(s)$ is polynomial in n, selecting the correct context $c \in C_\phi$ is intractable.

We assume no prior structure among the contexts that would allow efficient indexing or pruning. Therefore, any algorithm A attempting to decide Contextual-SAT must, in the worst case, query an exponential number of possible μ_c's. This makes both the verification and search procedures exponential in nature, violating the polynomial bound required for inclusion in **P**.

7 Philosophical Motivation

The foundation of this work draws on a perspectival-dialetheic framework in which propositions may be both true and false in different contexts until a collapse occurs. This aligns with certain forms of paraconsistent logic and contextualism in philosophy of language. [8, 9]

In our formulation, logical propositions are not evaluated absolutely but relative to an implicit frame of interpretation. This reflects real-world reasoning more faithfully than classical models and aligns with human semantic processing, where ambiguity and context are integral rather than pathological.

8 Limitations and Open Questions

While the argument shows that semantic ambiguity introduces an exponential barrier, several questions remain:

- Can intuitive structure be imposed and programmed for the context space C_ϕ to allow sub-exponential search, even though complexity is still increased by the search?
- What other domains can this framework be extended to?
- What are the implications for AI systems and models that must interpret ambiguous input if they are to mimic understanding?
- Should this motivate a change in classical logic itself?

These remain open and merit further investigation.

9 Appendix: Illustrative Examples

9.1 Solving the Liar Paradox

Consider the canonical Liar Sentence:

"This sentence is false."

In classical logic, this sentence causes oscillation:

$$L := \text{“This sentence is false”}, \quad v(L) = \texttt{true} \Rightarrow v(L) = \texttt{false}, \quad v(L) = \texttt{false} \Rightarrow v(L) = \texttt{true}$$

This behavior is paradoxical only under the assumption that: 1. The sentence expresses a classical proposition. 2. Its interpretation is context-independent.

Human Interpretation: A Lie About Itself

Now consider a context-sensitive semantic framework where sentences may be performative—that is, their truth depends on intent, not just structure.

Let context c evaluate the sentence pragmatically. The phrase "This sentence is false" is treated not as a paradox, but as a form of self-referential misrepresentation—a lie.

In ordinary language, we interpret:

"I am a liar" or *"Everything I say is false"*

as rhetorical or exaggerated—not logically invalid. Our contextual training leads us to extract the likely communicative intention: - The speaker is signaling dishonesty. - The content is not to be interpreted at face value.

This results in a contextual reinterpretation:

$$\text{“This sentence is false”} \quad \rightsquigarrow \quad \text{“This sentence should not be trusted”}$$

Which we semantically reframe as:

$$\mu_c(L) = \text{“I am lying”} \quad \Rightarrow \quad \mu_c(L) = \texttt{true}$$

The sentence, though logically structured as a negation of itself, functions as a **true statement about its own falseness**—and is understood as such through semantic realignment.

Semantic Reversal as Disambiguation

The apparent paradox disappears when we recognize that the contradiction arises only when we assume the sentence is telling the truth in a direct way. In the contextual frame c, the sentence is interpreted as a signal of deceit—thus its falsity becomes evidence of its deeper truth.

This reversal can be modeled semantically:

$$\text{If } \mu_c(L) = \texttt{false}, \text{ then the sentence is lying, which implies it is actually true.}$$

Or:

$$\mu_c(L) = \text{“This is a dishonest sentence.”}$$
$$\Rightarrow \text{“This sentence being false is the basis for calling it true.”}$$

Contextual-SAT Interpretation

Within the Contextual-SAT framework, such a sentence introduces a verification challenge: - The truth of L depends on an unmodeled context c that resolves its intended meaning. - Classical verification fails due to the self-referential negation. - Contextual disambiguation resolves the paradox via semantic realignment, yielding a coherent truth value.

Conclusion

This analysis reinforces a central thesis of our framework:

> Ambiguity in meaning can be computationally expensive to resolve, but contextual cues allow human and intelligent systems to bypass paradoxes through pragmatic interpretation.

Thus, paradoxes like the Liar are not undecidable per se; they are underconstrained.

9.2 Semantic Reinterpretation of an Alarm Formula

We illustrate the computational impact of semantic context using a simple CNF formula:

$$\phi = (x_1 \vee x_2) \wedge \neg x_3$$

This formula is syntactically satisfiable under any assignment where at least one of x_1 or x_2 is true and x_3 is false.

Let us consider a fixed assignment:

$$s = (x_1 = 0,\ x_2 = 1,\ x_3 = 0)$$

Context c_1: Emergency Training Simulation

- $x_1 =$ "danger"
- $x_2 =$ "safe"
- $x_3 =$ "false alarm"

Under this interpretation:

- $x_1 \vee x_2 = 0 \vee 1 = 1$
- $\neg x_3 = \neg 0 = 1$
- $\phi = 1 \wedge 1 = 1$

The interpretation is coherent: one signal indicates safety, the other indicates no real threat. The system concludes the situation is acceptable. Thus, the assignment satisfies ϕ under c_1.

Context c_2: Real-Time Emergency Protocol

- $x_1 =$ "alert issued"
- $x_2 =$ "alert confirmed"
- $x_3 =$ "valid alarm"

With the same assignment s, the syntactic logic still yields:

- $\phi = (0 \vee 1) \wedge \neg 0 = 1 \wedge 1 = 1$

However, semantically this may reflect a contradiction: both alert signals are present, but the alarm is *not* valid. In a system where any alert requires a valid alarm to be triggered, this interpretation is inconsistent or inadmissible. As such, the semantic evaluation function $\mu_{c_2}(\phi)(s)$ may reject the assignment—even though the Boolean logic allows it.

Interpretive Explosion

This small example reveals a key challenge:

> The truth value of ϕ is not invariant under reinterpretation of variables. The number of possible semantic contexts grows exponentially, and determining which $c \in C_\phi$ validates a given assignment becomes intractable.

Thus, even verifying a candidate solution s requires reasoning over exponentially many possible interpretations μ_c, violating the requirement that NP problems must have polynomial-time verifiers. This justifies the claim that Contextual-SAT resides in $\mathbf{NP} \setminus \mathbf{P}$.

10 Conclusion

We have constructed a decision problem, CONTEXTUAL-SAT, in which satisfiability depends on an unmodeled semantic context that cannot be extracted or verified in polynomial time. While verification under a fixed context is tractable, the absence of a globally knowable interpretive frame explodes the verification task into an exponential search over possible contexts. This places CONTEXTUAL-SAT squarely in $\mathbf{NP} \setminus \mathbf{P}$, thereby resolving the P vs NP question.

But more than a formal separation, this work exposes a category error at the heart of the classical P vs NP formulation: the assumption that verification is always a syntactic procedure. In reality, many verification tasks—especially those that emerge from language, reasoning, and modeling—carry inherent semantic ambiguity. These tasks are not just computational—they are epistemological. They ask not only "Is this string valid?" but "What does this string mean under the right interpretation?"

If we continue to treat inherently semantic uncertainty as a syntactic feature of computation, we will never resolve the P vs NP question. We will remain stuck in a closed loop, failing to verify what we cannot first interpret.

This is the same mistake that has allowed the Liar Paradox to persist for over two millennia. Its apparent contradiction only survives because humans—context-creating machines—have refused to lift it out of classical logic and place it in the epistemic-semantic domain where it belongs. The paradox was never a failure of logic, but a failure to recognize that some propositions don't submit to formal closure because they point beyond themselves.

We must not repeat this mistake with the P vs NP problem. The cost is not merely theoretical: the future of cryptography, artificial intelligence, and computational epistemology hinges on our ability to distinguish between formal solvability and semantic interpretability.

The separation of P and NP is not just a question of time complexity—it is a question of what computation truly is, and what it means to know that a solution has been found.

Conceptual Reasoning Engine Prototype for LLM Integration

```
import datetime
from enum import Enum
from collections import defaultdict
from typing import Callable, Dict, List, Tuple, Set
from dataclasses import dataclass, field
import ast
from functools import reduce

# ========== LOGIC ENUM ==========
class Tvalue(Enum):
FALSE = 0
TRUE = 1
BOTH = 2

def __invert__(self): # NOT
lookup = {
Tvalue.FALSE: Tvalue.TRUE,
Tvalue.TRUE: Tvalue.FALSE,
Tvalue.BOTH: Tvalue.BOTH
}
return lookup[self]

```

```
def __and__(self, other):
lookup = {
(Tvalue.TRUE, Tvalue.TRUE): Tvalue.TRUE,
(Tvalue.TRUE, Tvalue.BOTH): Tvalue.BOTH,
(Tvalue.BOTH, Tvalue.TRUE): Tvalue.BOTH,
(Tvalue.BOTH, Tvalue.BOTH): Tvalue.BOTH,
}
return lookup.get((self, other), Tvalue.FALSE)

def __or__(self, other):
lookup = {
(Tvalue.FALSE, Tvalue.FALSE): Tvalue.FALSE,
(Tvalue.FALSE, Tvalue.BOTH): Tvalue.BOTH,
(Tvalue.BOTH, Tvalue.FALSE): Tvalue.BOTH,
(Tvalue.BOTH, Tvalue.BOTH): Tvalue.BOTH,
}
return lookup.get((self, other), Tvalue.TRUE)

# ========== LOGICAL CONNECTIVES PARSER ==========
def parse_logical_expression(expr: str) -> Callable[[Dict[str, Tvalue]], Tvalue]:
expr_ast = ast.parse(expr, mode='eval')

def make_eval(node):
if isinstance(node, ast.BoolOp):
op = Tvalue.__and__ if isinstance(node.op, ast.And) else Tvalue.__or__
children = [make_eval(v) for v in node.values]
return lambda context: reduce(op, [child(context) for child in children])
elif isinstance(node, ast.UnaryOp) and isinstance(node.op, ast.Not):
child = make_eval(node.operand)
return lambda context: ~child(context)
elif isinstance(node, ast.Name):
return lambda context: context[node.id] if node.id in context else Tvalue.BOTH
else:
raise ValueError(f"Unsupported AST node: {type(node)}")

return make_eval(expr_ast.body)

# ========== PERSPECTIVE ==========
@dataclass
class Perspective:
name: str
evaluate_fn: Callable[[str], Tvalue]
memory: Dict[str, Tvalue] = field(default_factory=dict)

def evaluate(self, statement: str) -> Tvalue:
if statement not in self.memory:
self.memory[statement] = self.evaluate_fn(statement)
return self.memory[statement]

def merge_with(self, other: 'Perspective', new_name: str) -> 'Perspective':
def composed_eval(statement: str) -> Tvalue:
v1 = self.evaluate(statement)
v2 = other.evaluate(statement)
return v1 if v1 == v2 else Tvalue.BOTH
return Perspective(name=new_name, evaluate_fn=composed_eval)

# ========== COLLAPSE CACHE ==========
class CollapseCache:
def __init__(self):
```

```python
self.cache: Dict[str, List[Tuple[Tvalue, float]]] = defaultdict(list)

def record(self, prop: str, value: Tvalue, weight: float = 1.0):
self.cache[prop].append((value, weight))

def get_weighted_score(self, prop: str) -> float:
entries = self.cache.get(prop, [])
if not entries:
return 0.0
total_weight, net_score = 0.0, 0.0
for val, weight in entries:
if val == Tvalue.TRUE:
net_score += weight
elif val == Tvalue.FALSE:
net_score -= weight
total_weight += weight
return net_score / total_weight if total_weight > 0 else 0.0

def get_stability(self, prop: str) -> float:
return abs(self.get_weighted_score(prop))

# ========== PROPOSITION ==========
@dataclass
class Proposition:
statement: str
contextual_value: Tvalue = Tvalue.BOTH
perspective_values: Dict[str, Tvalue] = field(default_factory=dict)
collapse_history: List[Tuple[datetime.datetime, Tvalue]] = field(default_factory=
    list)

def _compute_dynamic_threshold(self, perspectives: List[Perspective]) -> float:
count = len(perspectives)
return min(max(0.5 / (1 + 0.1 * count), 0.1), 0.5)

def evaluate(self, perspectives: List[Perspective], cache: CollapseCache) ->
    Tvalue:
values: Set[Tvalue] = set()
for perspective in perspectives:
value = perspective.evaluate(self.statement)
self.perspective_values[perspective.name] = value
cache.record(self.statement, value)
values.add(value)

threshold = self._compute_dynamic_threshold(perspectives)
score = cache.get_weighted_score(self.statement)

if Tvalue.TRUE in values and Tvalue.FALSE in values:
self.contextual_value = Tvalue.BOTH
elif score > threshold:
self.contextual_value = Tvalue.TRUE
elif score < -threshold:
self.contextual_value = Tvalue.FALSE
else:
self.contextual_value = Tvalue.BOTH

self.collapse_history.append((datetime.datetime.now(), self.contextual_value))
return self.contextual_value

def compute_stability(self, cache: CollapseCache) -> float:
```

```
return cache.get_stability(self.statement)
```

What if a computer could reason about its own code

when it encountered a problem?

What if the answer to paradox is just making decisions?

Prototype Basic Halting Analyzer

```
import ast
import sqlite3
import logging
from enum import Enum
from typing import Callable, Dict, List, Tuple, Any
from dataclasses import dataclass, field
from collections import defaultdict
from concurrent.futures import ThreadPoolExecutor
import timeout_decorator

# === Logging ===
logging.basicConfig(filename="agent.log", level=logging.ERROR)

# === Truth Value Enum ===
class Tvalue(Enum):
FALSE = 0
TRUE = 1
BOTH = 2

# === Collapse Strategy ===
class CollapseStrategy:
def collapse(self, weighted_values: Dict[Tvalue, float], confidence: float) ->
    Tvalue:
p_true = weighted_values.get(Tvalue.TRUE, 0.0)
p_false = weighted_values.get(Tvalue.FALSE, 0.0)
threshold = 0.6 - 0.2 * (1.0 - confidence)
if p_true > threshold:
return Tvalue.TRUE
elif p_false > threshold:
return Tvalue.FALSE
return Tvalue.BOTH

# === Perspective Class ===
@dataclass
class Perspective:
name: str
epistemic_position: str
evaluate_fn: Callable[[str], Tvalue]
memory: Dict[str, Tvalue] = field(default_factory=dict)

def evaluate(self, statement: str) -> Tvalue:
if statement not in self.memory:
self.memory[statement] = self.evaluate_fn(statement)
return self.memory[statement]

# === Persistent Collapse Cache ===
class PersistentCollapseCache:
```

```
def __init__(self, db_path: str = "cache.db"):
self.conn = sqlite3.connect(db_path)
self.conn.execute("CREATE TABLE IF NOT EXISTS cache (prop TEXT, value TEXT, weight REAL)")
self.conn.commit()

def record(self, prop: str, value: Tvalue, weight: float = 1.0):
self.conn.execute("INSERT INTO cache (prop, value, weight) VALUES (?, ?, ?)",
(prop, value.name, weight))
self.conn.commit()

def get_weighted_score(self, prop: str) -> float:
cursor = self.conn.execute("SELECT value, weight FROM cache WHERE prop = ?", (prop,))
entries = [(Tvalue[val], weight) for val, weight in cursor.fetchall()]
if not entries:
return 0.0
total_weight = sum(weight for _, weight in entries)
net_score = sum((weight if val == Tvalue.TRUE else -weight) for val, weight in entries if val != Tvalue.BOTH)
return net_score / total_weight if total_weight else 0.0

def get_stability(self, prop: str) -> float:
return abs(self.get_weighted_score(prop))

# === Proposition Class ===
@dataclass
class Proposition:
statement: str
contextual_value: Tvalue = Tvalue.BOTH
perspective_values: Dict[str, Tvalue] = field(default_factory=dict)
confidence: float = 0.0

def evaluate(self, perspectives: List[Perspective], cache: PersistentCollapseCache,
weights: Dict[str, float] = None, strategy: CollapseStrategy = None) -> Tvalue:
weighted_values = defaultdict(float)
total_weight = 0.0
for p in perspectives:
value = p.evaluate(self.statement)
self.perspective_values[p.name] = value
weight = weights.get(p.name, 1.0) if weights else 1.0
cache.record(self.statement, value, weight)
weighted_values[value] += weight
total_weight += weight

if total_weight == 0:
self.contextual_value = Tvalue.BOTH
return self.contextual_value

probs = {k: v / total_weight for k, v in weighted_values.items()}
entropy = -sum(p * (1.0 / p) for p in probs.values() if p > 0)
self.confidence = 1.0 - (entropy / 1.585)
self.contextual_value = strategy.collapse(weighted_values, self.confidence)
return self.contextual_value

def compute_conflict_score(self) -> float:
values = set(self.perspective_values.values())
```

```
return 1.0 if Tvalue.TRUE in values and Tvalue.FALSE in values else 0.5 if Tvalue.
    BOTH in values else 0.0

# === Simulation Engine ===
class SimulationEngine:
def __init__(self, max_attempts=3):
self.loop_suspicions = defaultdict(int)
self.max_attempts = max_attempts

def analyze_ast(self, code: str) -> bool:
try:
tree = ast.parse(code)
for node in ast.walk(tree):
if isinstance(node, (ast.While, ast.For)):
if not any(isinstance(n, ast.Break) for n in ast.walk(node)):
return False
return True
except:
return False

@timeout_decorator.timeout(15)
def execute_code(self, code: str) -> bool:
try:
exec(code, {})
return True
except TimeoutError:
return False
except:
return True

def simulate(self, code: str) -> bool:
self.loop_suspicions[code] += 1
if self.loop_suspicions[code] >= self.max_attempts:
return False
if not self.analyze_ast(code):
return False
return self.execute_code(code)

# === Pattern Perspective ===
class PatternPerspective(Perspective):
def __init__(self):
super().__init__("pattern", "structural", self._pattern_eval)

def _pattern_eval(self, code: str) -> Tvalue:
if "while True" in code and "break" not in code:
return Tvalue.FALSE
if "return" in code and "if" in code:
return Tvalue.TRUE
return Tvalue.BOTH

# === Oracle Perspective (Decide Truth) ===
class OraclePerspective(Perspective):
def __init__(self):
super().__init__("oracle", "omniscient", self._oracle_eval)

def _oracle_eval(self, code: str) -> Tvalue:
return Tvalue.FALSE if "while True" in code else Tvalue.TRUE

# === Simulation Perspective (Dynamic Execution) ===
```

```
class SimulationPerspective(Perspective):
def __init__(self, engine: SimulationEngine):
self.engine = engine
super().__init__("simulation", "empirical", self._simulate)

def _simulate(self, code: str) -> Tvalue:
result = self.engine.simulate(code)
return Tvalue.TRUE if result else Tvalue.FALSE

# === Mock LLM Client ===
class MockLLMClient:
def query(self, prompt: str) -> dict:
if "while True" in prompt:
return {
"result": "FALSE",
"perspectives": [
{"name": "pessimistic", "reasoning": "detects loop"}
]
}
return {
"result": "TRUE",
"perspectives": [
{"name": "optimistic", "reasoning": "trusts structure"}
]
}

# === LLM Perspective ===
class LLMPerspective(Perspective):
def __init__(self, llm_client=None):
self.llm_client = llm_client or MockLLMClient()
super().__init__("llm", "semantic", self._llm_eval)

def _llm_eval(self, code: str) -> Tvalue:
response = self.llm_client.query(code)
return Tvalue[response["result"]]

# === Multi Perspective Agent ===
class MultiPerspectiveAgent:
def __init__(self, perspectives: List[Perspective], conflict_threshold=0.6):
self.perspectives = perspectives
self.cache = PersistentCollapseCache()
self.strategy = CollapseStrategy()
self.conflict_threshold = conflict_threshold
self.perspective_stability = {p.name: 1.0 for p in perspectives}

def evaluate_code(self, code: str) -> Tuple[Tvalue, Proposition]:
prop = Proposition(code)
with ThreadPoolExecutor() as executor:
futures = {p.name: executor.submit(p.evaluate, code) for p in self.perspectives}
for name, future in futures.items():
prop.perspective_values[name] = future.result()
value = prop.evaluate(self.perspectives, self.cache, self.perspective_stability,
    self.strategy)
return value, prop

# === Halting Solver ===
class HaltingSolver:
def __init__(self, llm_client=None):
self.llm_client = llm_client or MockLLMClient()
```

```
self.engine = SimulationEngine()

def solve(self, code: str) -> Tuple[bool, str]:
perspectives = [
PatternPerspective(),
OraclePerspective(),
SimulationPerspective(self.engine),
LLMPerspective(self.llm_client),
]
agent = MultiPerspectiveAgent(perspectives)
value, prop = agent.evaluate_code(code)
halts = value == Tvalue.TRUE
notes = f"Confidence: {prop.confidence:.2f}, Conflict: {prop.
    compute_conflict_score():.2f}"
return halts, notes
```

References

[1] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971.

[2] L. A. Levin. Universal search problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973.

[3] T. Baker, J. Gill, R. Solovay. Relativizations of the P = NP question. *SIAM Journal on Computing*, 1975.

[4] A. Razborov, S. Rudich. Natural proofs. *Journal of Computer and System Sciences*, 1997.

[5] K. Gödel. On formally undecidable propositions of Principia Mathematica and related systems. 1931.

[6] A. Tarski. The concept of truth in formalized languages. 1936.

[7] D. Jurafsky, J. Martin. *Speech and Language Processing*. Pearson, 3rd Ed.

[8] G. Priest. *In Contradiction: A Study of the Transconsistent*. 2nd ed., Oxford University Press, 2006.

[9] R. N. Giere. *Scientific Perspectivism*. University of Chicago Press, 2006.