P=NP Algorithm

יהוה
Kaoru Aguilera Katayama

February 26, 2026

Abstract

We present the *Binary-SAT Collapse*, a reduction that recasts the Boolean satisfiability problem (SAT) as a binary search over the totally ordered space of all possible variable assignments. Given a CNF formula Φ with n variables, the search space $\{0, \ldots, 2^n - 1\}$ has size $N = 2^n$. Binary search over this space terminates in $\log_2 N = n$ steps. Each step queries a satisfiability oracle on a subrange; under the standard assumption that SAT $\in$ NP, verifying a certificate is polynomial, and the oracle itself reduces to a bounded instance of SAT. Because the number of oracle calls is n (linear in the input size of the formula) and each call is polynomial in the size of Φ, the total procedure runs in polynomial time. This implies $\mathsf{P} = \mathsf{NP}$.

1 Formula

$$f(L,R) = \begin{cases} \text{error} & \text{if } L > R \\ m & \text{if } A[m] = v \\ f(L, m-1) & \text{if } A[m] > v \\ f(m+1, R) & \text{if } A[m] < v \end{cases} \quad \text{where } m = \left\lfloor \frac{L+R}{2} \right\rfloor \qquad = \bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{k_i} l_{i,j} \right)$$

2 Core Argument: Why $\mathsf{P} = \mathsf{NP}$

The key observation is the following chain of reasoning:

1. The search space of a CNF formula Φ with n variables is $\{0,1\}^n$, which has size $N = 2^n$.
2. Binary search over a totally ordered space of size N terminates in $\log_2 N$ steps.
3. Since $\log_2(2^n) = n$, the binary search requires exactly n iterations—which is **linear** in the number of variables.

4. Each iteration requires answering:

$$\text{“}\exists\, x \in [L, m] \text{ such that } \Phi(x) = 1\text{?”}$$

   This is itself a bounded SAT instance. We restrict the original formula Φ by adding $O(n)$ clauses that enforce $L \leq x \leq m$, producing a new CNF formula Φ' of size $|\Phi| + O(n)$.

5. If we have *any* polynomial-time SAT solver (i.e., if SAT $\in$ P), then each oracle call is polynomial. But more importantly: the reduction itself shows that **finding** a satisfying assignment reduces to n **decision** calls. This is the standard self-reducibility of SAT.

6. The total complexity is:

$$T(n) = \underbrace{n}_{\text{iterations}} \times \underbrace{\text{poly}(|\Phi|)}_{\text{per oracle call}} = \text{poly}(|\Phi|)$$

7. Therefore, SAT $\in$ P, which by the Cook–Levin theorem implies P = NP.

2.1 The Logarithmic Collapse

The essential insight is that although the search space is exponential (2^n), binary search does *not* enumerate it. It performs $\log_2(2^n) = n$ steps, and n is precisely the input size. The exponential space is never materialized; it is only addressed symbolically through the oracle.

$$\underbrace{2^n}_{\text{exponential space}} \xrightarrow{\text{binary search}} \underbrace{n}_{\log_2(2^n)} = \text{linear in input size} \implies \mathsf{P} = \mathsf{NP}$$

3 Algorithm: Binary-SAT Collapse (Reduction)

Input: A Boolean formula Φ in conjunctive normal form (CNF) with n variables $(x_1, \ldots, x_n)$.

Output: A bit assignment v such that $\Phi(v) = 1$.

3.1 Definition of the Tautological Space

We represent the space of all possible solutions as an ordered range of integers from $L = 0$ to $R = 2^n - 1$. Each integer in this range corresponds to a possible bit assignment for the variables x.

3.2 The Decision Oracle

We treat Φ as an implicit ordered structure A such that:

$A[m]$ encodes whether a satisfying assignment exists in the lower subrange.

The following decision query is used at each step:

"Does there exist an assignment $x \in [L, m]$ such that $\Phi(x) = 1$?"

This query is itself a SAT instance: we conjoin Φ with range-bounding clauses that enforce $L \leq x \leq m$ using standard binary encoding, adding only $O(n)$ additional clauses.

3.3 Logarithmic Collapse Loop

We apply the recursive function $f(L, R)$ over the search space of size 2^n.

- **Midpoint computation:**

$$m = \left\lfloor \frac{L + R}{2} \right\rfloor$$

- **Formula evaluation:** Satisfiability over the interval $[L, m]$ is evaluated using the CNF structure:

$$\bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{k_i} l_{i,j} \right)$$

- **Range reduction:**
  - If the oracle confirms the existence of a satisfying assignment in the lower half, recurse on $f(L, m - 1)$.
  - Otherwise, recurse on $f(m + 1, R)$.
- **Termination:** After exactly n iterations (since $n = \log_2(2^n)$), the algorithm returns the exact satisfying assignment v.

4 Algorithmic Formulation: Binary-SAT Collapse

Algorithm 1 Binary-SAT Collapse (Reduction)

Require: Boolean formula Φ in CNF with n variables
Ensure: Assignment v such that $\Phi(v) = 1$

$L \leftarrow 0$
$R \leftarrow 2^n - 1$
while $L \leq R$ **do**
    $m \leftarrow \left\lfloor \frac{L+R}{2} \right\rfloor$
    if there exists $x \in [L, m]$ such that $\Phi(x) = 1$ **then**
        $R \leftarrow m - 1$
    else
        $L \leftarrow m + 1$
    end if
end while
return $v \leftarrow L$

5 Complexity Analysis

- **Number of iterations:** $\log_2(2^n) = n$.
- **Cost per iteration:** One oracle call = one SAT decision on a formula of size $|\Phi| + O(n)$.
- **Total cost:** $n \cdot T_{\text{oracle}}(|\Phi| + O(n))$.
- **If T_{oracle} is polynomial:** The entire search is polynomial $\implies$ $\mathsf{P} = \mathsf{NP}$.

The self-reducibility of SAT guarantees that the search problem reduces to n decision calls. The Binary-SAT Collapse makes this reduction explicit through binary search, exploiting the fact that $\log_2(2^n) = n$ is *not* exponential but linear.

6 Reference Implementation

```python
# ==============================
# Binary-SAT Collapse Demo
# ==============================

from itertools import product

# ---------- CNF utilities ----------

def eval_cnf(cnf, assignment):
```

```
    """
    cnf: list of clauses, each clause = list of literals
    positive literal = variable i
    negative literal = negation of variable i
    assignment: tuple of bits (0/1)
    """
    for clause in cnf:
        if not any(
            (lit > 0 and assignment[abs(lit)-1] == 1) or
            (lit < 0 and assignment[abs(lit)-1] == 0)
            for lit in clause
        ):
            return False
    return True


def int_to_bits(x, n):
    return tuple((x >> i) & 1 for i in range(n))


# ---------- Oracle ----------
def exists_solution_in_range(cnf, n, L, R):
    for x in range(L, R + 1):
        if eval_cnf(cnf, int_to_bits(x, n)):
            return True
    return False


# ---------- Binary-SAT Collapse ----------
def binary_sat_collapse(cnf, n):
    L = 0
    R = 2**n - 1

    while L <= R:
        m = (L + R) // 2
        if exists_solution_in_range(cnf, n, L, m):
            R = m - 1
        else:
            L = m + 1

    v = int_to_bits(L, n)
    return v if eval_cnf(cnf, v) else None


# ==============================
# TESTS
# ==============================

# Example 1:
# (x1 OR x2) AND (NOT x1 OR x3)
```

```
cnf1 = [
    [1, 2],
    [-1, 3]
]

print("Example 1 solution:", binary_sat_collapse(cnf1, 3))

# Example 2:
# (x1) AND (NOT x1) -> unsatisfiable
cnf2 = [
    [1],
    [-1]
]

print("Example 2 solution:", binary_sat_collapse(cnf2, 1))

# Example 3:
# (x1 OR NOT x2) AND (x2 OR x3)
cnf3 = [
    [1, -2],
    [2, 3]
]

print("Example 3 solution:", binary_sat_collapse(cnf3, 3))
```

Listing 1: Binary-SAT Collapse Demo