Solving the Boolean k-SAT Problem in Polynomial-Time

Caleb P. Nwokocha

June 2025

Overview

I present a polynomial-time algorithm and its C implementation that finds the first satisfying assignment of satisfiable k-SAT instance in *lexicographic* (binary) order by carving out the *gaps* between forbidden patterns derived from each clause. Each clause of fixed length n induces a *forbidden bit-vector* $f_i \in \{0,1\}^n$: a '1' at position j if the jth literal is negative (forbidding bit-value 1), or '0' if positive. Converting these vectors to integers, sorting them, and enumerating the integer intervals between, before, and after them yields all assignments not excluded by any clause.

Key Steps:

1. Read, parse, and validate M clauses of n literals each (ensure uniform length, no duplicate literals, no v vs. $-v$ conflict).
2. For each clause, construct forbidden vector f_i of length n.
3. Convert each f_i to integer a_i via bit-shifts.
4. Sort the array $[a_1, \ldots, a_M]$ in ascending order using parallel quicksort.
5. Enumerate three regions:
   - *Head:* integers from 0 up to $a_1 - 1$,
   - *Gaps:* integers between each consecutive pair a_i, a_{i+1},
   - *Tail:* integers from $a_M + 1$ to $2^n - 1$.
6. Stop after printing the first integer if only the first satisfying assignment is desired; otherwise continue through all regions.

ALGORITHM

```
procedure SATGapEnumeration
    loop
        clauses ← ReadClauses()
        if conflict detected in clauses then
            print "Unsatisfiable: immediate conflict"
            continue
        end if
        M ← |clauses|, n ← literals per clause
        Build forbidden vectors f_1, ..., f_M
        a[i] ← VecToInt(f_i) for i = 1 ... M
        ParallelQuicksort(a, 1, M)
        printed ← false
                                                        ▷ Head enumeration
        if a[1] > 0 then
            for v = 0 to a[1] − 1 do
                PrintBits(v, n); printed = true
                break              ▷ stop if only first assignment is desired
            end for
        end if
                                                        ▷ Gap enumeration
        for i = 1 to M − 1 do
            if a[i + 1] > a[i] + 1 then
                for v = a[i] + 1 to a[i + 1] − 1 do
                    PrintBits(v, n); printed = true
                    break
                end for
            end if
        end for
                                                        ▷ Tail enumeration
        if a[M] < 2^n − 1 then
            for v = a[M] + 1 to 2^n − 1 do
                PrintBits(v, n); printed = true
                break
            end for
        end if
        if ¬printed then
            print "Unsatisfiable: no satisfying assignment"
        end if
        break
    end loop
end procedure
```

Time Complexity Analysis

To find only the *first* satisfying assignment, I incorporate the early-exit mechanism. Let M be the number of clauses and n the clause length.

1. Reading, Parsing, and Validation

- Tokenization: $O(M, n)$ to scan all clauses.
- Duplicate checks: $O(n^2)$ per clause, total $O(M, n^2)$.
- Conflict detection: $O(n^2)$ per clause, total $O(M, n^2)$.

Cumulative: $O(M, n^2)$.

2. Forbidden Vector Construction: $O(M, n)$ for one pass per clause.

3. Integer Conversion: $O(M, n)$ bit-shifts over n bits for M clauses.

4. Parallel quicksort: $O(M \log M)$ work, $O(1)$ comparisons.

5. Early-exit Enumeration: At most one region yields the first assignment:

- Head: If $a_1 > 0$, one comparison then print $v = 0$ in $O(n)$, exit. Total $O(n)$.
- Gaps: In worst case, examine up to $M - 1$ intervals ($O(M)$ checks). Upon finding $a_{i+1} > a_i + 1$, print one assignment in $O(n)$, exit. Total $O(M + n)$.
- Tail: One comparison ($a_M < 2^n - 1$), then print in $O(n)$, exit. Total $O(n)$.

Worst enumeration cost: $O(M + n)$. Best: $O(n)$.

Summing all contributions from parsing/validation, sorting, and early-exit enumeration gives

$$T_{\text{first}}(M, n) = O(M\, n^2) \; + \; O(M \log M) \; + \; O(M + n).$$

Because for moderate n the $O(M\, n^2)$ term usually dominates $O(M \log M)$ and $O(M + n)$, I simplify this to

$$T_{\text{first}}(M, n) = O\big(M\, n^2 + M \log M\big).$$

Implications on P vs NP

The program ability to find the first satisfying assignment in polynomial time does suggest that $P = NP$. The Boolean Satisfiability Problem (SAT) is a classic NP-complete problem, meaning that it is as hard as the hardest problems in NP. While it is generally assumed that finding any satisfying assignment (or solving SAT in general) is hard, this is based on the complexity of solving SAT instances completely. However, the algorithm discussed in this paper only finds one satisfying assignment and operates in polynomial time, which challenges the traditional view of SAT computational hardness.

The key point here is that finding any satisfying assignment is sufficient to show that the formula is satisfiable. If this task can be completed in polynomial time, then SAT is solvable in polynomial time, and therefore, $P = NP$. The focus on the first satisfying assignment is not a trivial modification; it is, in fact, an essential part of solving the SAT problem. In essence, finding one solution is equivalent to solving the problem in the decision version of SAT.

The Search for a Single Solution

At the heart of NP-completeness lies the challenge of searching through a vast solution space. For NP-complete problems, such as SAT, this search is thought to require exponential time in the worst case. However, if a polynomial-time algorithm can find even one solution (the first satisfying assignment), it implies that the search space for finding that solution is much smaller than previously thought.

This result challenges the assumption that NP-complete problems, in general, require exponential time for solution search. Finding one solution is sufficient to show that a problem is solvable, and if that solution can be found efficiently, the entire class of NP problems becomes subject to polynomial-time algorithms.

Implications for P = NP

- Polynomial-time solution for k-SAT: The algorithm discussed in this article can find the first satisfying assignment in polynomial time. This is a direct polynomial-time solution for k-SAT, which is an NP-complete problem. Therefore, this result suggests that k-SAT can be solved in polynomial time, which implies that $P = NP$.

- Broader implications: If k-SAT can be solved in polynomial time for finding any satisfying assignment, it is likely that other NP-complete problems (such as Vertex Cover, Traveling Salesman Problem, etc.) can also be solved in polynomial time. This would imply that the entire NP class can be solved in polynomial time, further reinforcing the possibility that $P = NP$.