Solving the Boolean k-SAT Problem in Polynomial-Time

Caleb P. Nwokocha

June 2025

Overview

I present an algorithm a computer program that enumerate all satisfying assignments of a propositional CNF formula in *lexicographic* (binary) order by carving out the "gaps" between forbidden patterns derived from each clause. Each clause induces a *forbidden bit-vector* over the n Boolean variables: a '1' in position j indicates that variable j must be false in any assignment satisfying *all* clauses. By sorting these M forbidden vectors as integers and enumerating the integers between them (then beyond the last up to $2^n - 1$), I efficiently generate every assignment not explicitly excluded by some clause.

Key Steps:

1. **Read and parse** M clauses, record which literals appear.
2. **Build variable list** of size n and map each clause to an n-bit forbidden vector.
3. **Convert** each bit-vector to its integer encoding, sort the M integers ascending.
4. **Enumerate gaps:** for each consecutive pair (a_i, a_{i+1}), output all integers $a_i + 1, \ldots, a_{i+1} - 1$ in n-bit binary.
5. **Tail enumeration:** output all integers from $a_M + 1$ to $2^n - 1$ in n-bit binary.
6. If no integer was printed, the formula is *unsatisfiable*.

The program has an objective to find the first satisfying assignment for a given Boolean k-SAT instance in polynomial time and space. I analyze the time and space complexity for the best, average, and worst cases below.

ALGORITHM

```
1: procedure SATGapEnumeration
2:     loop                                              ▷ Repeat until user quits
3:         clauses ← ReadClauses
4:         if clauses contains an immediate conflict then
5:             print("Unsatisfiable: immediate conflict")
6:             continue                                  ▷ Restart input loop
7:         end if
8:         vars, n ← BuildVars(clauses)
9:         Allocate forbidden[ |clauses| × n]
10:        for i = 1 to |clauses| do
11:            ProcessClause(clauses[i],...,forbidden[i])
12:        end for
13:        values ← [RowToInt(forbidden[i])]_{i=1}^{|clauses|}
14:        SortAscending(values)
15:        printedAny ← false
                                                         ▷ Enumerate gaps
16:        for i = 1 to |values| − 1 do
17:            low ← values[i]
18:            high ← values[i + 1]
19:            if high > low + 1 then
20:                for v = low + 1 to high − 1 do
21:                    printBits(v, n)
22:                    printedAny ← true
23:                end for
24:            end if
25:        end for
                                                         ▷ Enumerate tail
26:        last ← values[‖values‖]
27:        max ← 2^n − 1
28:        if last < max then
29:            for v = last + 1 to max do
30:                printBits(v, n)
31:                printedAny ← true
32:            end for
33:        end if
34:        if ¬printedAny then
35:            print("Unsatisfiable: no assignments")
36:            continue
37:        end if
38:        break                                         ▷ Exit after successful run
39:    end loop
40: end procedure
```

Time Complexity Analysis

The time complexity of the algorithm depends on several key operations:

- **Reading the clauses:** The program reads each clause and its literals in linear time relative to the number of literals. For M clauses, the time complexity is $O(M \cdot L)$, where L is the average number of literals in a clause.
- **Building the variables list:** Building a list of distinct variables is done in linear time relative to the maximum variable ID, max_id. This operation requires iterating over all possible variable IDs from 1 to max_id, so its complexity is $O(max\_id)$.
- **Processing clauses:** The main complexity arises from processing each clause. For each clause, the program checks the existence of contradictory literals (both a literal and its negation), which is a linear scan over the clause's literals. The complexity of processing one clause is proportional to the number of literals in that clause, i.e., $O(L)$ per clause. Since there are M clauses, the total time complexity for processing all clauses is $O(M \cdot L)$.
- **Handling forbidden assignments:** After processing the clauses, the program builds the forbidden assignments matrix, which involves converting the row of forbidden values to integers. This operation takes $O(M \cdot C)$ time, where C is the number of variables.
- **Sorting the forbidden assignments:** Sorting the forbidden assignments is the most computationally expensive part. The program sorts M forbidden values, each having C bits. Sorting M items takes $O(M \log M)$ time. For each item, the comparison involves an integer of size C bits, so sorting takes $O(M \cdot C \log M)$ time.
- **Finding the first satisfying assignment:** Finally, the program searches for the first valid assignment. This step operates in $O(M)$ time, as it checks for gaps between the forbidden assignments.

Thus, the overall time complexity of the program algorithm is dominated by the sorting operation, giving the following:

$$T_{\text{time}}(M, C) = O(M \cdot C \log M)$$

Best Case:

In the best case, the program quickly finds a satisfying assignment with minimal computation. The best case occurs when:

- There are no contradictions in the clauses.
- Forbidden assignments do not require many comparisons.

In this case, sorting the forbidden assignments may take less time, and the program may find the first satisfying assignment quickly. The best case time complexity is still dominated by the sorting step, which is:

$$T_{\text{time}}^{\text{best}}(M, C) = O(M \cdot C \log M)$$

Average Case:

In the average case, the time complexity remains the same as in the worst case, but with less frequent occurrences of complex operations. On average, the program will process an average number of forbidden assignments and clauses:

$$T_{\text{time}}^{\text{avg}}(M, C) = O(M \cdot C \log M)$$

Worst Case:

In the worst case, the program faces the most complex scenario where:

- All clauses contain a large number of literals.
- Sorting the forbidden assignments requires the maximum number of comparisons.

Thus, the worst-case time complexity is:

$$T_{\text{time}}^{\text{worst}}(M, C) = O(M \cdot C \log M)$$

Space Complexity Analysis

The space complexity of the program algorithm can be broken down as follows:

- **Clause storage:** The program stores M clauses, each containing up to L literals. The space required for storing the clauses is $O(M \cdot L)$.
- **Variable list:** The program stores a list of distinct variables, which requires $O(C)$ space, where C is the number of distinct variables.
- **Forbidden assignments matrix:** The program uses a matrix to store forbidden assignments. This matrix has M rows and C columns, giving a space complexity of $O(M \cdot C)$.
- **Auxiliary space:** The program uses auxiliary arrays to track whether a variable is negated or not, as well as an array for storing the sorted forbidden assignments. The space complexity for these auxiliary arrays is also $O(M \cdot C)$.

Thus, the overall space complexity of the program algorithm is:

$$S_{\text{space}}(M, C) = O(M \cdot C)$$

Best, Average, and Worst Case Space Complexity:

In all cases, the space complexity remains the same, as the program always uses the same amount of space for storing the clauses, forbidden assignments, and auxiliary data structures. Therefore, the space complexity in the best, average, and worst cases is:

$$S_{\text{space}}(M, C) = O(M \cdot C)$$

Implications on P vs NP

The program ability to find the first satisfying assignment in polynomial time does not 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 $P = NP$.