Potential Polynomial Time Algorithm For 3-SAT

Author: Vedaang Devendra Pagnis
Affiliation: Department of Computer Engineering and Information Technology,
Veermata Jijabai Technological Institute
ORC ID : https://orcid.org/0009-0001-2237-2708

***Abstract*—This paper presents a heuristic polynomial-time algorithm for the 3-SAT problem, motivated by contributions to the longstanding P versus NP question. While the algorithm is incomplete and does not solve all instances of 3-SAT, it demonstrates significant practical strengths. Notably, it achieves high-quality approximate solutions efficiently and shows competitive performance against established solvers such as MiniSAT and WalkSAT. The paper includes a detailed pseudocode description of the algorithm, a formal proof of its polynomial time complexity, comprehensive empirical evaluation, and comparative analysis. Finally, potential directions for further research and improvement are discussed.**

I. Introduction

For completeness and proper context, we will provide a brief background on the classes of problems.

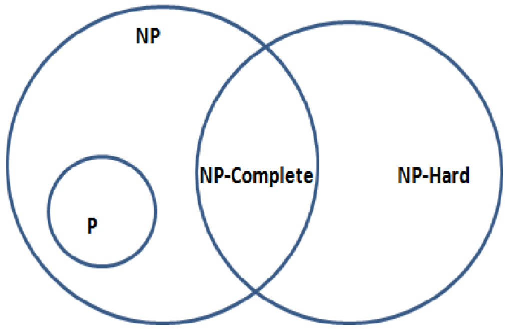
P problems - The set of problems which can be solved using a deterministic and polynomial time algorithm. Examples of such problems include : searching, sorting, graph traversal etc.

NP problems - The set of problems that are not solvable in polynomial time, but given a solution(called 'a certificate'), the solution can be verified in polynomial time.

NP-Hard problems - The set of problems such that any problem in NP can be reduced to these problems in polynomial time. However, these problems themselves may not be in NP : there are NP-Hard problems that are neither solvable nor verifiable in polynomial time(as per our current knowledge). Hence, they are "harder" than NP problems.

NP-Complete problems - The set of problems that are verifiable in polynomial time but may or may not be solvable in polynomial time. However, every problem in NP can be converted to a problem in NP-Complete in polynomial time. Thus, these problems are in both, NP and NP-Hard.

The below image provides a graphical view of the classes. Our focus is on NP-Complete problems.



Graphically, the P versus NP problem is to determine whether the set NP is the same as P. This would imply that the graph would reduce to three sets : P, NP-Hard and NP-Complete as an intersection of P and NP-Hard, because all problems in NP (and hence, those in NP-Complete too) would then become P.

What is the 3-SAT Problem?

The 3-SAT problem is a variant of the SAT problem. The SAT problem is an NP-Complete problem. SAT stands for "satisfiability".

SAT Problem : Given a Boolean expression containing variables(x1,x2,..) and connectives(AND,OR,NOT), is there a possible assignment of values(0 or 1) to the variables such that the expression evaluates to 1?

It is NP-Complete because a certificate can be verified in polynomial time(we can put the values in the expression and evaluate), and it is also as hard as any other problem in NP because no polynomial time algorithm is yet known for this problem.

A variant of this problem called the "3-SAT" problem exists wherein the expression is given in the form of a CNF(conjunctive normal form). Further, every clause has only three variables in it. The 3-SAT problem can be shown to be NP-Complete.

We claim that we have found a potential polynomial time algorithm for the 3-SAT problem. It is not yet complete : it does not solve all instances of the 3-SAT. We prove formally that the algorithm has a very good virtue : it is polynomial in time. Moreover, we conducted empirical testing on SATLIB instances and we found some very promising results. For the uf20 instances, (with each clause having 20 variables and 91 clauses), the algorithm satisfies 475 instances out of the total 1000 instances(47.5% of the dataset) completely by finding a solution in polynomial time, indicating a non-trivial property of the algorithm.

II. Problem Statement

Given a 3-SAT instance, is it possible to solve it in polynomial time?

The challenge is to not just solve the 3-SAT problem but to solve it efficiently. Based on the current knowledge that we have, we propose the following algorithm.

III. Methodology

We now focus on the main algorithm. Below is a brief description of the algorithm, followed by a pseudocode.

1) Initialise a dictionary wherein the keys are the variables and the values will be a set of the clauses in which the variables have appeared. Note that, we store a variable and its negation separately(for example, x1 and ~x1 are stored as separate keys).
   Initialise a multiset which will store the satisfied clauses.
   Initialise a set which will store the set variables.
2) Start traversing the given 3-SAT instance, clause by clause. Whenever a variable is encountered, add the current clause number to the value(set) of the variable in the dictionary.
3) After all clauses have been traversed completely, sort the dictionary according to the cardinality of the value set in non-increasing order. Thus, the variables with the maximum number of occurrences will be selected first.
4) Start setting the variables in order of the sorted dictionary. If the variable's value set has all clauses which are already satisfied, continue without setting the variable. If the variable satisfies at-least one unsatisfied clause till now, set it.
   However, before setting a variable, check whether its negation is also set. If the negation is set, check the length of the value sets of both variables. Set the variable which has the longer length.
   Remove the clauses of the earlier set variable(if any) and add the clauses of the newly set variable, into the multiset. Accordingly, also change the set of "set variables".
5) Before each iteration(before each new variable is set), check the "satisfied clauses" set. If it contains all the clauses(that is, all clauses satisfied), return the set. Otherwise, continue with the further checks.
6) Continue until all variables(or their respective negations) are set. This is the end of the first pass.
7) In the second pass, create a set of unsatisfied clauses. For each unsatisfied clause, select a variable from the clause. Check the clauses satisfied by the negation(the negation would be set, otherwise by contradiction the current clause should have been satisfied). If there are multiple occurrences of all the clauses satisfied by the negation of the considered variable in the multi-set, the current variable can be set. Add the current clause to the set of satisfied clauses.
8) Continue this process until the number of unsatisfied clauses reduce to zero or there is no other possible reconfiguration of the variables.

IV. Pseudocode

```
FUNCTION SAT_Solver(clauses, variables):

    // 1. Initialization
    assignment ← random True/False values for each variable
    max_satisfied_count ← count_satisfied_clauses(clauses, assignment)
    assignment_steps ← empty list

    // 2. First Pass: Greedy Improvement Loop
    WHILE True:
        best_flip_var_candidate ← None
        best_flip_val_candidate ← None
        best_candidate_net_gain ← -∞
        best_candidate_multiset_cost ← ∞

        FOR each clause IN clauses:
            IF clause is not satisfied under assignment:
                FOR each variable IN clause:
                    // Consider flipping variable's value
                    flipped_assignment ← copy of assignment
                    flipped_assignment[variable] ← NOT assignment[variable]
            // Calculate net gain = (# newly satisfied clauses) - (# newly unsatisfied clauses)
            net_gain ← calculate_net_gain(clauses, assignment, flipped_assignment)

            // Calculate multiset cost heuristic for this flip (tie-breaker)
            multiset_cost ← calculate_multiset_cost(variable, flipped_assignment)

            // Update best candidate if better net gain or better cost on tie
            IF (net_gain > best_candidate_net_gain) OR
               (net_gain == best_candidate_net_gain AND multiset_cost < best_candidate_multiset_cost):
                best_flip_var_candidate ← variable
                best_flip_val_candidate ← flipped_assignment[variable]
                best_candidate_net_gain ← net_gain
                best_candidate_multiset_cost ← multiset_cost

        // Apply the best flip found in this iteration if beneficial
        IF best_flip_var_candidate IS NOT None AND best_candidate_net_gain > 0:
            assignment[best_flip_var_candidate] ← best_flip_val_candidate
            max_satisfied_count ← count_satisfied_clauses(clauses, assignment)

            // Log step
            assignment_steps.APPEND((
                best_flip_var_candidate,
                best_flip_val_candidate,
                "First pass flip with net gain: " + best_candidate_net_gain +
                ", multiset cost: " + best_candidate_multiset_cost
            ))
        ELSE:
            // No improvement found, exit loop
            BREAK
```

```
    // 3. Second Pass: Targeted Clause Improvement Loop
    second_pass_steps ← empty list
    FOR iteration FROM 1 TO MAX_ITERATIONS:
        // Select a target clause that is currently unsatisfied
        target_clause, target_clause_idx ← select_unsatisfied_clause(clauses, assignment)
        IF target_clause IS None:
            // All clauses satisfied
            BREAK

        best_flip_var_candidate ← None
        best_flip_val_candidate ← None
        best_candidate_net_gain ← -∞
        best_candidate_multiset_cost ← ∞

        FOR each variable IN target_clause:
            flipped_assignment ← copy of assignment
            flipped_assignment[variable] ← NOT assignment[variable]

            net_gain ← calculate_net_gain(clauses, assignment, flipped_assignment)
            multiset_cost ← calculate_multiset_cost(variable, flipped_assignment)

            IF (net_gain > best_candidate_net_gain) OR
               (net_gain == best_candidate_net_gain AND multiset_cost < best_candidate_multiset_cost):
                best_flip_var_candidate ← variable
                best_flip_val_candidate ← flipped_assignment[variable]
                best_candidate_net_gain ← net_gain
                best_candidate_multiset_cost ← multiset_cost

        // Apply the best flip candidate only if:
        //  (a) it strictly improves satisfaction (net_gain > 0), OR
        //  (b) it maintains satisfaction count (net_gain == 0) AND
        //      it satisfies the target clause (previously unsatisfied)
        IF best_flip_var_candidate IS NOT None AND (
            best_candidate_net_gain > 0 OR
            (best_candidate_net_gain == 0 AND
             NOT is_clause_satisfied(target_clause,
assignment) AND
             is_clause_satisfied(target_clause,
{assignment with best_flip_var_candidate
flipped}))
        ):
            assignment[best_flip_var_candidate] ←
best_flip_val_candidate
            max_satisfied_count ←
count_satisfied_clauses(clauses, assignment)

            second_pass_steps.APPEND((
                best_flip_var_candidate,
                best_flip_val_candidate,
                "Second pass flip for target clause " +
target_clause_idx +
                " with net gain: " +
best_candidate_net_gain +
                ", multiset cost: " +
best_candidate_multiset_cost
            ))
        // else no beneficial flip found, proceed to next
iteration

    // 4. Final evaluation
    final_satisfied_count ←
count_satisfied_clauses(clauses, assignment)

    IF final_satisfied_count == LENGTH(clauses):
        RETURN True, assignment,
final_satisfied_count, assignment_steps +
second_pass_steps
    ELSE:
        RETURN False, None,
final_satisfied_count, assignment_steps +
second_pass_steps
```

V. Complexity Analysis

Let the instance contain ‘m’ clauses and ‘n’ variables.
The algorithm contains some key steps such as :

1) Initialising the dictionaries of clauses. This requires a complete traversal of the clauses. Thus, the time complexity of this step = 3*m.
2) Sorting of the dictionary is done based on the length of the value set. Hence, this step requires n* log n time when an algorithm such as Heap sort or Timsort is used.
   Further, some time will be required to traverse the set of the variables in the dictionaries.

   Note : We claim the upper bound on this time as :
   n*log(n) + required time <= n*log(n) + m*n
   This is because each of the ‘n’ variables can occur at-most in each of ‘m’ clauses(this case will never be feasible - by the definition of 3-SAT - one clause has exactly 3 variables. But this calculation provides a convenient and absolute upper bound of the time required here).
   Time complexity = n*log(n) + m*n

3) Setting the variables involves a lookup operation of its negation. The lookup operation involves O(1)(constant = c) time because it is a set lookup. Further, if both are not set(that is, the variable or its negation is being visited for the first time), then we compare the cardinalities of the value set of the variable and its negation. This will involve a time of at-most 2*m(because the size of the value set of both variables will never exceed m).
   Thus, the time complexity for this step = (2*m + c)

4) Checking if the length of satisfied clauses set is equal to the given number of clauses requires constant time. If yes, the algorithm returns the set of variables with their values. If not, the algorithm returns the count of the satisfied clauses. Thus, the data structures being evaluated already : returning requires constant time.

Keeping all these time complexities in mind, the overall time complexity of the algorithm is :

<u>Time complexity</u> = 3*m + n*log(n) + m*n + 2*m + k
= (n+5)*m + n*log(n) + k
= O(n*m)

The above is for the first pass.
For the second pass, we focus on the unsatisfied clauses. Considering them to be ‘u’ in number, in the worst-case : every variable from each unsatisfied clause needs to be checked. Hence, all 3 variables(literals) from each clause have to be checked, contributing to 3*u time. Although ‘u’ is guaranteed to be lesser than ‘m’, for simplicity we consider the time of the second pass as “3*m” which is the upper bound. Thus, the total time of both passes are :

Time complexity = c.n*m(first pass) + 3*m(second pass)
= O(n*m)

= Polynomial time complexity

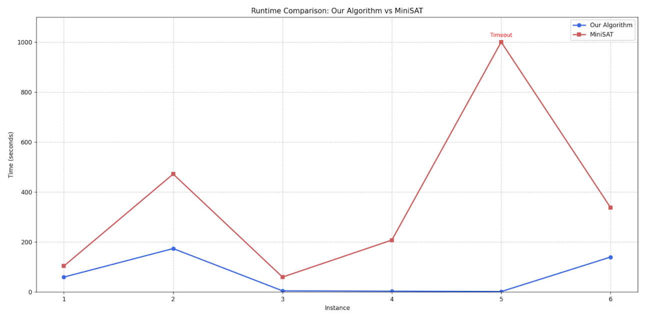
VI. Testing And Claims

The algorithm has been tested extensively on various benchmarks and instances, ranging from SATLIB instances to instances from the SAT Competition. The following are the individual performance results of the algorithm :

1) We tested the algorithm on all 1000 instances of the uf20 SATLIB dataset. Each instance contains 20 variables and 91 clauses. In this dataset, 475 instances were completely satisfied by the algorithm, by giving a correct solution. In the remaining 525 instances, the average number of clauses satisfied by the algorithm's solution are 90/91. Thus, the algorithm performs exceptionally well for uf20 : it can efficiently solve instances in polynomial time.
2) The algorithm was tested on 100 instances each of the uf50 and uf250 SATLIB benchmarks. In the uf50 instances, the algorithm found a satisfying assignment for 16 instances but none for the uf250. Nevertheless, the average clause satisfaction is 216/218 in remaining instances of uf50 and 1056/1065 in uf250, which still indicate high quality of solutions.
3) We tested the algorithm on controlled backbone sized datasets, with the backbone size ranging from 10 to 90 out of 100 variables. However, in these instances we consistently received solutions that satisfied 99%+ clauses. Hence, one important claim is that the algorithm is robust to backbone sizes which is important because backbone size is one of the most dominant parameters for algorithm performance.

To demonstrate the utility of the algorithm, we compared the performance of the algorithm with "MiniSAT", a state-of-the-art solver. For this we employed SAT Competition instances which are industrial scale instances : the smallest instance comprising 1,62,928 variables and 14,23,778 clauses and the largest containing 27,05,815 variables and 3,31,81,429 clauses. We executed both the algorithms on 6 such instances, and plotted a graph of the time taken(in seconds) for each instance tested.

The graph is as follows :



The red plot is for MiniSAT while the blue plot is for our algorithm. From the graph, one can easily make out that our algorithm is significantly better than MiniSAT.
On one instance, MiniSAT could not give a solution at all(timeout). Moreover, the algorithm maintains its satisfaction rate above 98% for these instances, with just one instance falling to 93%. We present an important deduction regarding our algorithm :

1) If an instance is satisfiable, our algorithm is not guaranteed to return the exact solution, but as testing has confirmed : it is capable of giving near-optimal variable assignments in a very short time. Hence, it is better than MiniSAT with respect to time : for an industrial instance, our algorithm gives an almost perfect solution in time which differs by a large order of magnitude practically.
2) If an instance is unsatisfiable, our algorithm is guaranteed to return it as unsatisfiable(it never claims false positives) and also gives a near-optimal solution to satisfy a maximum number of clauses. Hence, industries in the realm of OR or decision-making who devise conditions/clauses without considering satisfiability, can use our algorithm's solution to perform some mitigation in case of such situations. MiniSAT only returns UNSATISFIABLE in such cases and does not give an assignment, hence it does not directly help in optimisation.

Considering these caveats and facts, we claim that we have conclusive evidence to show that our algorithm outperforms MiniSAT.

VII. Theoretical Foundations

We now present the theoretical framework for the algorithm. Our initial motivation for SAT was to view it as a combinatorial optimisation problem, hence we investigated combinatorial structures that are closely related to such problems.

There are some important observations that prompted the design of such an algorithm, which include the following :

1) SAT is inherently about the number of clauses satisfied. The solution to a SAT instance is a set which contains the assignments of all variables as either "True" or "False".
2) A solution set to a SAT instance always starts with an empty set, because at the start, none of the variables are assigned anything. Furthermore, this empty set is a subset(trivially) of the main solution set.
3) A solution set is built level-by-level from the empty set by adding the set of assigned variables at each step.

There is a combinatorial structure called the "greedoid". A greedoid is a tuple (E, G), such that :
E is a set and G is a set of "feasible" subsets of E such that the following properties are satisfied :

1) Accessibility : For every feasible set A, there exists an element "a" belonging to A, such that A - {a} is also feasible.
2) Augmentation : If A and B are both feasible and |A| > |B|, then there exists an element "a" belonging to A - B, such that B U {a} is also feasible.

Considering the above observations and the properties of a greedoid, we have the following lemma :

Lemma 1 : SAT instances can be modelled as greedoids

Let E be the solution set of a given SAT instance. Initially, E is phi. As the algorithm makes its first assignment x1, E U {x1} is the new set. As the algorithm progresses, the assignments added further satisfy more clauses(due to the first-pass strategy). Thus, we define the "feasibility" in the context of a SAT instance most naturally as a solution set that satisfies a subset of the clauses and does not cause any conflicts in variables.

Under this model, given a set A which is feasible(at some stage of the algorithm). Thus, this set was formed by the algorithm due to the addition of a variable assignment. The previous set itself had to be feasible because otherwise it would not have been considered by the algorithm(if there are conflicts, the algorithm does not consider those cases). Hence, the accessibility property is followed.

Moreover, if A and B are two feasible sets at different levels then the elements belonging to A - B would be variable assignments. Hence, the set B U {a} for an "a" belonging to the difference set is feasible because the set A is built incrementally from B(note that : there may be some variables introduced after conflict resolution in the second pass in A, but these variables when added to B, form a feasible set).

Thus, the accessibility and augmentation properties are followed. Thus, (E, G) is a greedoid.

Greedoids are generalisations of matroids, where greedy algorithms are guaranteed to work. Hence, our algorithm is inspired by the greedoid structure of SAT that is shown here. The theoretical foundations serve as a proof or a strong indicator of why our algorithm performs so well because it is tapping on these properties. However, it fails on cases also, indicating that while the SAT follows this property - the algorithm might have some shortcomings.

So far, our approach has the perspective of an optimisation problem. However, SAT can also be viewed as a decision problem : does a solution exist so that all clauses are satisfied? This leads to insights into "Oracle" machines which are a specific type of Turing machines.
However, an oracle has a property that whatever results it gives(a decision made by the oracle) is correct for that instance. In more formal terms, an oracle is a machine that guarantees the following(in the context of SAT) :

1) If it says "satisfiable", it means that the instance is actually satisfiable.
2) If it says "unsatisfiable", it means that the instance is actually unsatisfiable.

Property 1 is said to be the positive oracle property while property 2 is said to be the negative oracle property. It is fairly straightforward to see that our algorithm satisfies property 1. However, it does not always satisfy property 2(there are instances which are satisfiable but our algorithm claims them as unsatisfiable).

VIII. Limitations And Future Scope

1) The algorithm is not complete : it does not solve all instances of the 3-SAT. However, it showcases the ability to provide highly effective approximations to 3-SAT instances in general. Even though the algorithm is not theoretically deterministic, it can be used in fields such as Machine Learning, to solve 3-SAT instances more efficiently than common heuristics.
2) More work can be done for pushing the algorithm towards completeness. As our work has demonstrated a fact that more memory can outweigh some time, we think that more judicious use of memory(example : incorporation of some cyclic dependency detection and resolution mechanism) could possibly lead to completeness, while retaining polynomial complexity. This would imply P=NP