B. TECH. PROJECT REPORT

On

BOOLEAN SATISFIABILITY

BY

Rohith Bellamkonda

DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY INDORE

May 2022

BOOLEAN SATISFIABILITY

A PROJECT REPORT
Submitted in partial fulfillment of the
requirements for the award of the degrees
of
BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE AND ENGINEERING

Submitted by:
Rohith Bellamkonda
(150001004)

Guided by:
Dr. Narendra S Chaudhari

INDIAN INSTITUTE OF TECHNOLOGY INDORE
May 2022

CANDIDATE'S DECLARATION

We hereby declare that the project entitled **"Boolean satisfiability"** submitted in partial fulfillment for the award of the degree of Bachelor of Technology in 'Computer Science and Engineering' completed under the supervision of **Dr. Narendra S Chaudhari from the Department of computer science and engineering,** IIT Indore is an authentic work.

Further, I/we declare that I/we have not submitted this work for the award of any other degree elsewhere.

Signature and name of the student(s) with date

CERTIFICATE by BTP Guide(s)

It is certified that the above statement made by the students is correct to the best of my/our knowledge.

Prof Narendra Chaudhari (Supervisor) 26 May 2022

Signature of BTP Guide(s) with dates and their designation

Preface

This report on “Boolean Satisfiability" is prepared under the guidance of Dr. Narendra S Chaudhari from the Department of Computer Science and Engineering, IIT Indore.

Through this report I have tried to implement an SAT solver using the Davis–Putnam–Logemann–Loveland (DPLL) algorithm. The DPLL algorithm-based Satisfiability solvers uses backtracking. In the early 1960s, two seminal articles introduced the basic search strategy, which is today known as the Davis–Putnam–Logemann–Loveland algorithm ("DPLL" or "DLL"). Recently many SAT solving approaches have been derived from and have same structure as the DPLL algorithm. They frequently only increase the efficiency of certain types of problems on SAT, like those used in industrial applications or those created randomly. The DPLL family of algorithms has been shown to have exponential lower bounds theoretically.

Rohith Bellamkonda
B.Tech. IV Year
Discipline of Computer Science and Engineering
IIT Indore

Acknowledgements

First and foremost, we want to convey our gratitude to our research supervisor, Dr. Narendra S Chaudhary whose contribution to stimulating suggestions and encouragement, helped us to coordinate our B. Tech project.

It is their help and support, due to which we were able to complete the design and technical report.

Without their support this report would not have been possible.

Rohith Bellamkonda
B.Tech. IV Year
Discipline of Computer Science and Engineering
IIT Indore

Abstract

I have tried to implement an SAT solver using the Davis–Putnam–Logemann–Loveland (DPLL) algorithm. The DPLL algorithm consists of three stages, Unit Propagation, Pure literal Elimination and backtracking by choosing a literal which occurs most frequently in all the clauses combined. I used ReactJS to make the user Interface for inputting the clauses and the internal code takes the input and performs the unit propagation, pure literal Elimination and backtracking and provides the output stating whether the given Boolean formula in conjunctive normal form is satisfiable (there exists a result of true for various combination of Boolean values) or unsatisfiable (when various combination of Boolean values of the given literal fails to provide a true value as a result)

Table of Contents

1 INTRODUCTION

SAT solver - A SAT solver is a computer program that seeks to solve the Boolean satisfiability problem in computer science and formal techniques. A SAT solver outputs whether a formula over Boolean variables, such as "(a or b)" and "(a or not b), is satisfiable, indicating that there are conceivable values of a and b that make the formula true, or unsatisfiable, meaning that there are no such values of a and b. The solution should return "satisfiable" in this example since the formula is satisfiable when a is true. Modern SAT solvers have evolved into complicated software artefacts involving a significant number of heuristics and program optimizations to work efficiently since the advent of SAT algorithms in the 1960s.

Converting a formula to conjunctive normal form is a common first step for SAT solvers. They are frequently built on fundamental algorithms like the DPLL algorithm, but they include a variety of modifications and features. Most SAT solvers have time-outs, so they'll finish in a reasonable amount of time even if they can't find a solution and return "unknown."

DPLL SAT solver – Davis–Putnam–Logemann–Loveland (DPLL) algorithm. The DPLL algorithm-based Satisfiability solvers uses backtracking. In the early 1960s, two seminal articles introduced the basic search strategy, which is today known as the Davis–Putnam–Logemann–Loveland algorithm ("DPLL" or "DLL"). Recently many SAT solving approaches have been derived from and have same structure as the DPLL algorithm. They frequently only increase the efficiency of certain types of problems on SAT, like those used in industrial applications or those created randomly. The DPLL family of algorithms has been shown to have exponential lower bounds theoretically.

2 DPLL ALGORITHM

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm is a complete, backtracking-based search technique for determining the satisfiability of propositional logic formulae in conjunctive normal form.

It was developed by Martin Davis, Donald W. Loveland and George Logemann, in 1961 as a modification of the older Davis–Putnam algorithm, which was devised by Davis and Hilary Putnam in 1960 as a resolution-based approach. The Davis–Logemann–Loveland algorithm is also known as the "Davis–Putnam method" or the "DP algorithm" in older literature. DLL and DPLL are two more frequent names that keep the distinction.

The normal backtracking method works by picking a random literal and assigning a Boolean value to it and then recursively simplifying the formula, and then verifying whether the simplified or modified formula is satisfiable; if so, the initial formula is satisfiable; if not, assume the negation of the truth value and then again check recursively. The dividing rule divides the issue into two subproblems. All true classes in the assignment are removed from the formula.

2.1 The working process of DPLL

True or False is returned by DPLL(F) when given a CNF formula F and the procedure is given below:

• Use unit propagation for as long as you can:

All sentences containing l should be removed from F and ~l from the remaining clauses to get a unit clause of the type Ci = l.

• Use as much pure literal elimination as possible:

Remove any clauses that include a literal l that only appears only either positively or negatively

• Return true if F doesn't have any clauses or F is empty.

Return false if F includes the empty clause.

• Splitting: If not, choose a literal l and make Val(l) Equal t. All sentences containing l should be removed from F and ~l from the remaining clauses to reduce F to F1.

Calculate the DPLL (F1). Return true if the result is true.

• If not, set val(l) = f. All sentences containing l should be removed from F and ~l from the remaining clauses to reduce F to F2. Return the result of DPLL(F2) calculation.

The picking of branching literal, which is the literal examined during backtracking, is critical to the Davis-Logemann-Loveland algorithm. As a result, this is more of a family of algorithms, one for each conceivable branching literal choice. The choice of branching literal has a significant impact on efficiency: depending on the branching literals used, the execution time might be either constant or exponential.

On average, DPLL is relatively fast; scenarios where a bad literal choice is the cause of exponential runtime are quite rare. However, there exist formulae in which any approach for choosing the branching variable results in an exponential runtime.

Algorithm 1: DPLL algorithm (implemented in `sat.py`)

```
Result: Either SAT or UNSAT
while not all variables assigned do
    if unit propagation returns conflict then
        return UNSAT
    end
    x ← choose splitting var
    val ← choose initial assignment for x
    create new decision level with x=val
    while unit propagation returns conflict do
        if decision level == 0 then
            return UNSAT
        end
        backtrack and set splitting variable for previous decision level to be the negation of
         the original choice
    end
end
return SAT
```

3 SAT HARD

Every problem that a machine with an infinite number of parallel threads can answer in polynomial time may be represented as an SAT problem. So, if you can answer the SAT quickly,

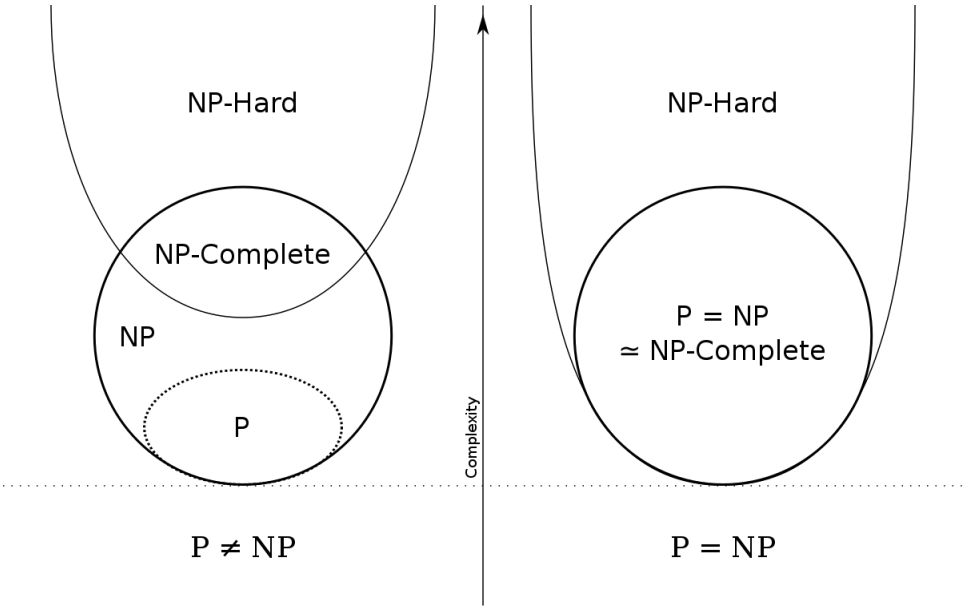
on a typical machine you can rapidly answer all of these questions. Theoreticians call this property of SAT NP-completeness because they use a nondeterministic machine model to represent unbounded parallelism – something we won't be able to do until quantum computers are available – and have demonstrated that SAT is the most difficult problem that such a machine can solve. While it is impossible to prove that all SAT solvers in worst case must use exponential time, thousands of problems for which the best-known algorithm is exponential are "easier" than SAT.

3.1 NP-HARD

A problem X is said to be in NP-Hard if it can be reduced to X in polynomial time by an NP-Complete problem Y. NP-Complete issues are just as difficult as NP-Hard problems. The problem does not have to be in the NP class to be NP-Hard.

3.2 NP-COMPLETE

If there is an NP problem Y that can be reduced to X in polynomial time, then the problem X is NP-Complete. NP-Complete problems are just as difficult as NP-Complete problems. If an issue is part of both NP and NP-Hard Problems, it is NP-Complete. In polynomial time, a non-deterministic Turing computer can solve an NP-Complete problem.



	P	NP	NP-complete	NP-hard
Solvable in polynomial time	✓			
Solution verifiable in polynomial time	✓	✓	✓	
Reduces any NP problem in polynomial time			✓	✓

4 MAKE DPLL EFFICIENT

The fact that every problem that can be solved with a Non-Deterministic Turing Machine in polynomial time can be encoded as a Conjunctive Normal Form formula that is SAT if and only if the Non-Deterministic Turing Machine "accepts" the given input string indicates that SAT is a difficult problem to solve.

Many individuals have worked on refining the fundamental DPLL algorithm and have achieved incredible improvements without altering the core concept.

Work on enhancing the algorithm is now being done in four ways.

(1) Do not replicate the formula.

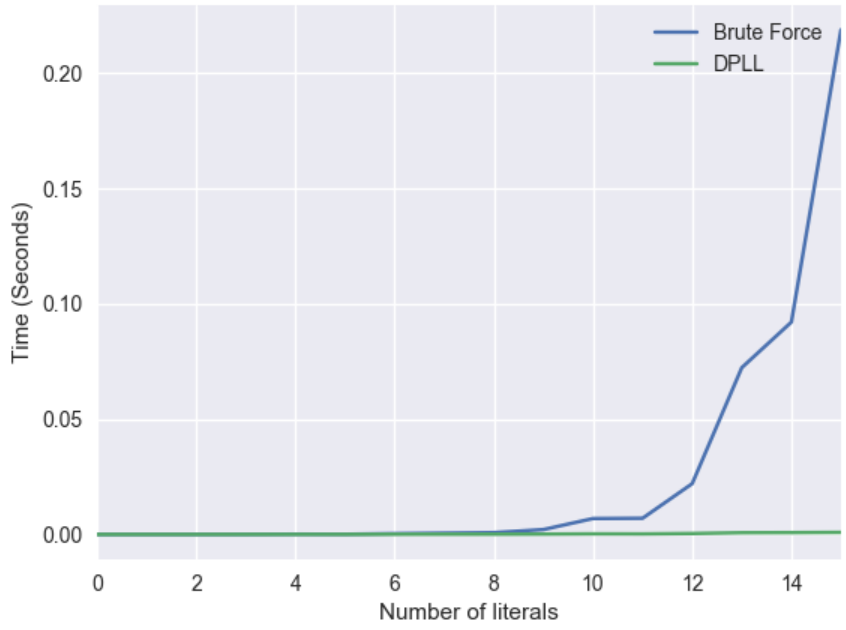
(2) By using new data structures to speed up the method, particularly the unit propagation section.

(3) establishing alternative branching literal selection policies

(4) describing several versions of the fundamental backtracking method

All types of optimizations are feasible in all three domains if one uses familiarity with efficient techniques in the appropriate fields.

4.1 DPLL VS BRUTEFORCE



4.2 PARTIAL VALUATIONS

Literals must be deleted from a formula at several phases of the DPLL technique. While this looks obvious, dealing with thousands of clauses makes this impossible. we utilize partial valuations, in which some but not all of the variables in a formula are assigned truth values.

As a result, we simply change the value of literal to f instead of removing the literal from the clause. Actually, this literal is only needed during splitting and unit propagation, we'll have kept ~l to t, thus there won't be anything to do.

This eliminates several needless processes.

As a result, the concept of a unit clause must be redefined. A unit clause, rather than having only one literal, has all but f is assigned to any of its literals. This is the same as removing the literals, but there is no need for formula changing.

4.3 USING DATA STRUCTURES

Counting the amount of these literals in each clause by going through the entire formula each time makes no sense. However, because there will be millions of clauses, keeping a record of the count of literals in every sentence that aren't false will be impossible.

The so-called watched literals method, which goes like this, is a superior approach. We choose two unassigned literals for each clause C in F to keep an eye on.

Then we need two lists for every variable x in F, one of sentences where x is monitored, and another of clauses where x is monitored.

When the value t is assigned to x, in the watch list for check all clauses.

– Check for another variable y in that sentence that may be watched and add that clause to y's watch list. We have a unit clause if everything but one literal l in the phrase is assigned f, and we may assign t to l and recur

– Continue if all literals have previously been allocated true

– if f has been assigned to every literal in the clause (through another op), F is unsat.

Even though this appears to be more complex initially, it considerably minimizes the clauses count that must be reviewed every time.

Watch lists do not need to be restored when the algorithm wants to travel backwards.

4.4 CHOOSING BRANCH LITERALS RULES

Choosing the proper literal to split has a significant impact on the runtime of algorithm. Developers develop heuristics, but they must ensure that calculating the heuristic is not so costly.

So, variables with high frequency count are chosen.

4.5 BASIC BACKTRACKING ALGORITHM VARIANTS

The naïve algorithm frequently divides and investigates in considerable detail to discover that a given Boolean formula is unsatisfiable. It again goes to the previous variable and does backtracking and fails to meet the Boolean formula once more. The rationale was the same as before, but the algorithm was unaware of this. To enhance backtracking, one must be able to reuse data gained in a previous branch.

Clause learning is the core method: if a clause that has conflicts is detected, construct a conflict clause comprising all variables that are currently termed false. Return to the first decision level, when one of these literals was still to be allocated. I'll stop there since the technicalities are a little tough, but the benefits are fantastic if you do it correctly, because we prevent a lot of wasted time looking for a satisfactory valuation.

4.6 ACHIEVING SIGNIFICANT PROGRESS

There are further low-level enhancements that can be made, resulting in SAT solvers that can now handle millions of literals. There exist some times when these processes underperformed on minor formulae, but generally, they perform admirably.

"Conjunctive Normal Form" should be used for our proposal.

Before we begin, we must verify that the proposition we are working with is in CNF.

It truly refers to a proposal that is full with v and ∧. So, basically, there should be no arrows. Your parenthesis-separated sub-clauses should be separated by instead of. You should take the following actions to do this:

1. Replace (1 ⇔ 2) with (1 ⇒ 2) ∧ (2 ⇒ 1)
2. Replace (1 ⇒ 2) with (~1 ∨ 2)
3. Move negations inside
   a) ~(~1) =1
   b) ~ (1 ∧ 2) = ~1 ∨ ~2
   c) ~ (1 ∨ 2) = ~1 ∧ ~2
4. Distribute ∨ over ∧:
   1 ∨ (2 ∧ 3) = (1 ∨ 2) ∧ (2 ∨ 3)

5 UNIT PROPOGATION

The technique is based on unit clauses, which are conjunctive normal form clauses made up of a single literal. If a group of clauses includes the unit clause, the subsequent clauses are simplified using the following two rules:

1) Each clause containing l is eliminated

2) This literal gets removed from every sentence that contains ~l.

When both rules are combined, new clauses are created that are identical to the previous one.

Let's have an example, the example contains unit clause 1, and perform unit propagation

{1 V 2, ~1 V 2, ~3 V 4, 1}

This sentence 1 V 2 will be eliminated entirely because 1 V 2 contains 1. This literal -1 can be omitted from the sentence since ¬1∨3 carries ~1. Unit clause 1 is retained; otherwise, the resultant set would not be identical to the original; however, this clause can be omitted if the data is already saved in another format. The following is a summary of the effect of unit propagation:

{1 V 2, ~1 V 2, ~3 V 4, 1} => {, 2, ~3 V 4,1}

The collection of clauses {3, ¬3∨4,1} coming from this is equal to the one above. The new unit clause 3 can be used as unit literal to perform unit propagation again

5.1 USING A PARTIAL MODEL

A partial model can be used to store or produce unit clauses that are present in a set of clauses. In this instance, unit propagation is done using the partial model's literals, and unit clauses are removed if their literal occurs in the model. In the example above, to the partial model unit clause 1 would be added; the simplification of the set of clauses would then proceed as previously, with the exception that unit clause 1 would be eliminated. The generated collection of clauses is identical to the original one, assuming that the partial model's literals are valid.

5.2 COMPLEXITY OF UNIT PROPOGATION

The total set she to check, which is sum of the sizes of all clauses, where each clause's size is the number of literals it contains, necessitates time quadratic in the total size of the set to check.

Unit propagation, on the other hand, may be done in linear time by preserving the list of clauses containing each literal for each variable.

This fundamental data structure may be built in a time proportionate to the size of the set, and it makes finding all sentences containing a variable fairly simple. For all unit clauses the overall running time for performing unit propagation is proportional to the size of the collection.

6 PURE LITERALS

Pure literal elimination is another feature of the original DPLL algorithm. Pure literal elimination is a method of locating and assigning literals that only have one polarity within the clauses that have not yet been satisfied. These literals are safe to employ since they do not falsify clauses. However, this isn't employed in the CDCL SAT solvers' search algorithm since recognizing pure literals is too costly for the benefit it provides.

If “a” does not appear "~a" in F, literal "a" is pure in a CNF formula. Pure literals may always be set to true without impacting satisfiability, thereby deleting the classes that contain them. Because this may cause additional literals to become pure, the procedure must be repeated to create a satisfiability equivalent formula that does not contain any pure literals. This is referred to as "pure literal elimination."

Many satisfiability algorithms employ the deletion of pure literals as a heuristic. It is still used by DLL-type algorithms that obtain the best theoretical worst-case upper limits for the CDCL-SAT solver implementations presently employ a data structure that is optimized for unit propagation, sacrificing the pure literal heuristic, whereas few recent solvers still use the heuristic only in their preparation step.

7 BACKTRACKING

Backtracking is an algorithmic strategy for recursively solving problems by attempting to develop a solution progressively, one piece at a time, and discarding any solutions that do not fulfil the problem's criteria at any point in time Backtracking is likewise a step up from the brute-force technique. So, the concept behind the backtracking approach is that it looks for a solution to a problem among all the possibilities accessible.

We begin the backtracking process by selecting one feasible choice, and if the problem is solved with that option, we return to the solution; otherwise, we retrace and choose another option from the remaining alternatives. There may also be instances where none of the possibilities provide a solution, in which case we recognize that retracing will not provide a solution. Backtracking can also be considered a type of recursion. This is due to the fact that the process of determining a solution from the numerous options available is continued recursively until we either find a

solution or reach the end state. As a result, we may infer that retracing at each stage removes options that are unlikely to lead to a solution and instead focuses on options that are likely to lead to a solution.

When it comes to backtracking, there are three categories of issues to consider:

We look for a feasible solution to this decision problem.

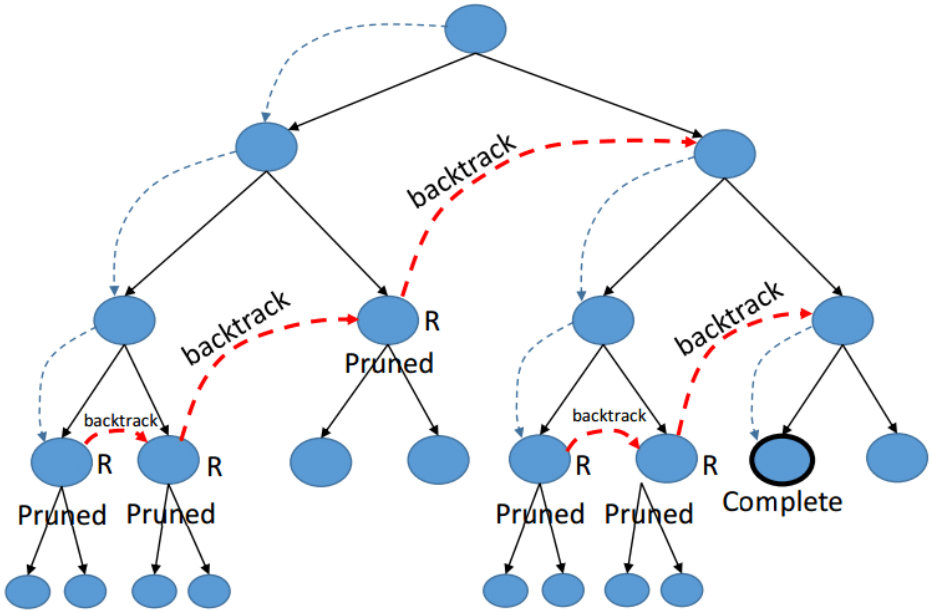
We seek the optimal solution to an optimization problem.

We find all possible solutions to the enumeration problem.

7.1 HOW TO CHECK WHETHER A PROBLEM CAN BE SOLVED USING BACKTRACKING?

Backtracking can be used to solve any constraint satisfaction problem with clear and well-defined constraints on any objective solution, which incrementally builds candidates to the solution and abandons them ("backtracks") when it determines that they cannot possibly be completed to a valid solution. Well-known algorithms such as DP or Greedy Algorithms can outperform the backtracking technique in every way. However, there are still a few issues that can only be solved via backtracking techniques.

```
procedure EXPLORE(node n)
    if REJECT(n) then return
    if COMPLETE(n) then
        OUTPUT(n)
    for n_i : CHILDREN(n) do EXPLORE(n_i)
```



7.2 RECURSIVE BACKTRACKING

Following these two simplification phases, the DPLL must choose a variable to branch on. The satisfiability problem is then divided into two parts: if the formula is satisfiable with the selected variable set to true or false, and whether the formula is satisfiable with the chosen variable set to true or false. In essence, the method "guesses" a variable to be true, then checks if that subproblem is satisfiable recursively; if it isn't, the programmer "guesses" the variable to be false and tries again.

You have complete freedom in deciding which variables to branch on. You might choose a variable at random, choose the variable that appears the most in the formula, or just take the alphabetically next variable. We define "reasonable" as a procedure that produces accurate outcomes.

If you don't put "involves all the original set of variables " in the check below, DPLL will only return a partial instance, which may exclude values for variables that were eliminated during the simplification process. We need to remember the initial set of variables because we're updating the formula recursively.

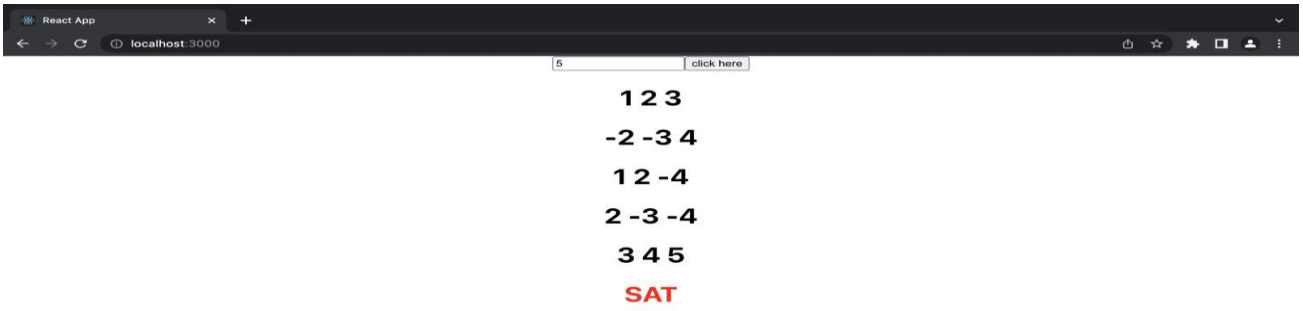
8 DATASETS

The datasets for testing the program have been taken from various online sources and compared the results of my sat solver to the already existing sat solvers on the internet.

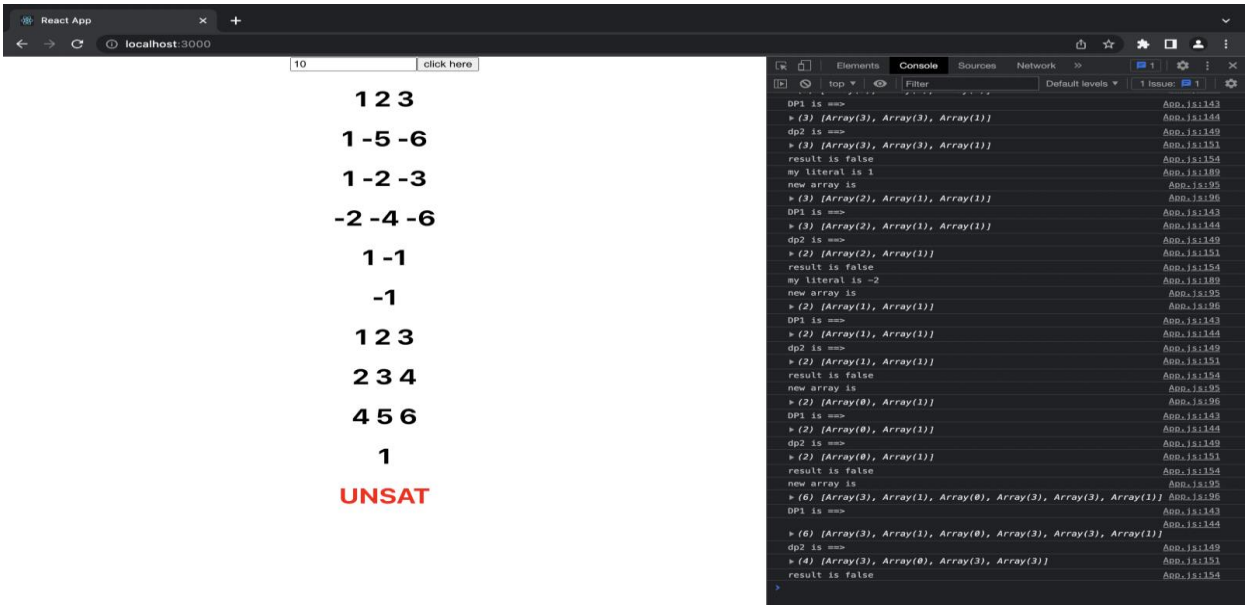
The dataset which I checked consists of one thousand satisfiable Boolean formulas and one thousand unsatisfiable Boolean formulas and this dataset is taken from GitHub. The link for datasets has been provided in the references section.

9 RESULTS

When a given Boolean formula is satisfiable, the result will be SAT



When a given Boolean formula is unsatisfiable, the result will be UNSAT



10 CONCLUSIONS AND SCOPE FOR FUTURE WORK

I have implemented a SAT solver using the DPLL algorithm.

The model now only tells whether the given Boolean formula is satisfiable or unsatisfiable but not the combination of truth values of the literals which makes the formula satisfiable if the formula is a satisfiable one.

I want to make a SAT solver website by providing a good interface and to get the result by just uploading the data in file instead of typing or pasting the data.

I also want to implement and present the backtracking tree of the given conjunctive normal form on user Interface for better understanding of the DPLL algorithm.

REFERENCES

1. https://davefernig.com/2018/05/07/solving-sat-in-python/
2. https://www.geeksforgeeks.org/backtracking-introduction/
3. https://www.cs.cornell.edu/courses/cs4860/2009sp/lec-04.pdf
4. https://jix.one/refactoring-varisat-3-cdcl/
5. https://www.cs.upc.edu/~erodri/webpage/cps/theory/sat/DPLL/slides.pdf
6. https://www.cs.cmu.edu/~15414/f17/lectures/10-dpll.pdf
7. http://www.diag.uniroma1.it//~liberato/ar/dpll/dpll.html
8. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.380.2837&rep=rep1&type=pdf
9. https://cs.stackexchange.com/questions/9556/what-is-the-definition-of-p-np-np-complete-and-np-hard
10. https://en.wikipedia.org/wiki/DPLL_algorithm
11. https://en.wikipedia.org/wiki/Unit_propagation
12. https://ptgmedia.pearsoncmg.com/images/9780134397603/samplepages/9780134397603.pdf
13. https://www-cs-faculty.stanford.edu/~knuth/news.html
14. https://cs.stackexchange.com/questions/9556/what-is-the-definition-of-p-np-np-complete-and-np-hard
15. https://github.com/khamkarajinkya/Davis-Putnam-Logemann-Loveland-Algorithm/blob/master/sat_tests.zip