EduSAT: Theory and Applications of Boolean Satisfiability

Yiqi Zhao , Ziyan An
School of Engineering (of Vanderbilt University), Nashville
yiqi.zhao@vanderbilt.edu

Abstract

Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) are widely used in automated verification. EduSAT is a framework created for educational purpose for SAT and SMT solving. In this project, we explore the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and the Reduced Order Binary Decision Diagram (ROBDD) for SAT solving. We also create an SMT solver operating on integer predicates as an extension of the SAT solvers. We provide solver abstractions based on the SMT solver for five NP-complete problems other than SAT and SMT. Apart from these, we also provide thorough documentations and testings as well as a SAT parser and a tree-based SAT formula generator. In our evaluation, we find 100% accuracy for all of the SAT and the SMT solvers. The DPLL solver with heuristics outperform the naive solver without heuristics in all cases of the evaluation for the DPLL SAT solver.

Index Terms

satisfiability, NP-completeness, graph coloring, subset sum

I. Introduction

Formal verification is a critical field in Computer Science with a focus to prove or disprove the safety, liveness, and other properties of a system using mathematical techniques such as induction. It is widely applied to circuit design [1], autonomous vehicles [2], and operating systems [3], etc. Many solutions to verification problems relate to the solutions of Boolean Satisfiability Problems (SAT) and its variant, Satisfiability Modulo Theories (SMT). Bounded Model Checking (BMC) of a discrete system, for instance, can be encoded as an equivalent SAT problem and solved with a domain specific solver such as Z3 [4]. Furthermore, by the Cook-Levin Theorem, SAT is NP-complete, which suggests that problems that are NP, such as graph coloring, can be reduced to a SAT problem by a deterministic Turing Machine in polynomial time.

Despite the significance of SAT and SMT, we observe a lack of open-source solvers for SAT problems with a pedagogical purpose not to mention demonstrations of how some classical NP-complete problems are encoded into SAT formulations. With this motivation, we introduce EduSAT, a Boolean Satisfiability Solver with support for SMT over integer predicates and extensions for solvers of 5 NP-complete problems (apart from SMT and SAT). In the solver, we showcase both the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and the ROBDD (reduced order binary decision diagram) and how they are used in solving SAT problems. The solver is crafted with a pedagogical purpose and is stored in a github repository with detailed tutorial (we will make the repository public with the permission of the instructor from CS 6315 and conduct further optimization and/or evaluations to improve its pedagogical utility). However, in this report, we focus on the technical content of the solver rather than evaluations of its educational implications, because this project is part of CS 6315, Automated Verification, where we need to demonstrate a proficiency in exercising the techniques learned from the course.

We list our contributions below:

1) We design a DPLL SAT solver and a ROBDD SAT solver (permitting single and all solutions) and allow clients to compare the performances of DPLL with the naive tabular methods for SAT. To the best of our knowledge, there is no previous work with open-source ROBDD implementations of a SAT solver.
2) We use recursive backtracking to extend the SAT solvers to an SMT solver over integer predicates.
3) We evaluate the performance of the SAT solvers on both their accuracies and efficiencies and we further evaluate the accuracy of the SMT solver through case studies. With the evaluations, we analyze the effects of number of variables, number of formulas, depth, and if multiple solutions are needed on the efficiency of the DPLL SAT solver through controlled experiments.
4) We construct solver abstractions for 5 NP-complete problems using the SMT solver crafted. We provide examples to how these abstractions can be used.
5) We organize the framework in a private repository and create detailed documentations on how such framework can be used as well as the technical details of the framework, which we will submit to brightspace together with this report.

We organize the rest of the report as follows: In section II, we discuss some related work. In section III, we introduce the technical details of the SAT solvers, the SMT solver, and how the SMT solver is used to solve NP-complete problems. In section IV, we evaluate our SAT and SMT solvers and discuss the performances. In section V, we summarize the limitation of EduSAT, and finally in VI we conclude the report.

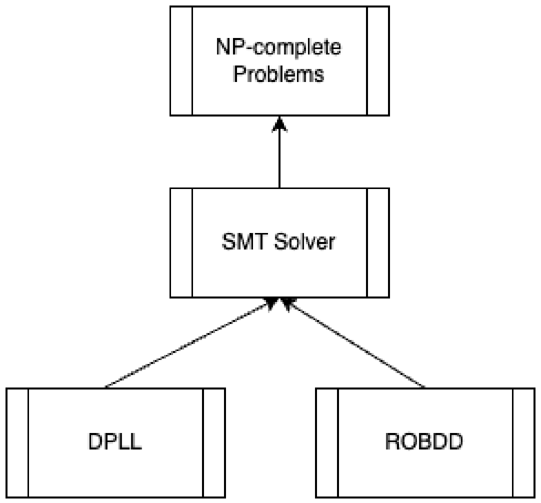

Fig. 1: Internal Structure of EduSAT (The arrows are not used for inheritance.)

II. Related Work

a) SMT solvers: One of the most commonly used SMT solver is Z3 [4], which is based on DPLL and incorporates quantifier instatiation using E-matching and relevancy propagation, etc. Despite that the codes from Z3 are available for public use, Z3 is designed for efficient solutions in real-life circumstances without a focus on the educational value. Despite that we also use DPLL in our work, we limit the DPLL algorithm to SAT solver only without a DPLL(T) extension for SMT. Other state-of-the-art SMT solvers include CVC4 (Cooperating Validity Checker) [5], which is also based on DPLL(T) and uses simplex method for its arithmetic, and Yices 2 [6], which uses a SAT solver with CDCL approach. To the best of our knowledge, none of the models above offers a user-controllable comparison with the naive tabular method and there is no educational tool that extends the solvers above to solving classical NP-complete problems other than SAT and SMT.

b) Solutions to NP-complete problems: Many NP-complete problems can be directly solved with an SMT solver. The map coloring problem, as suggested by [7], can be encoded into SMT constraints where no neighboring regions have the same color. We adopt the same strategy to our graph coloring solver presented in III. However, some NP-complete problems presented in this report do not have to depend on SMT solvers. For instance, to solve the Subset Sum problem, presented in III, we can directly utilize dynamic programming. We offer the solver abstrations with SMT to introduce a demonstration of the flexibility of SMT in encoding and solving complex problems.

III. Methods

A. SAT and SMT

Boolean Satisfiability (SAT) refers to the problem to determine if there exists a model to a Boolean formula P such that the model evaluates to true. SMT extends the acceptable formula, P to many-sorted formulas. In EduSAT, we only consider formulas with integer variables with the following operators: $+, -, \times, //, >, <, \leq, \geq, =$, where $//$ represents integer division.

To give a concrete example of a SAT problem, $x0 \vee x1$ is satisfiable with a candidate model of $x0 = 1, x1 = 1$. On the contrary, $x0 \wedge \neg x0$ is not satisfiable. An example of an SMT problem is $(y0 + 3 < 5) \wedge (y1 - 1 \geq 3)$, which is satisfiable with a candidate model of $y0 = 1, y1 = 5$. On the contrary, $(y1 > 1) \wedge (y1 = 1)$ is unsatisfiable.

B. Overview

The internal structure of EduSAT, as presented in Figure 1, consists of two SAT solvers, DPLL algorithm and ROBDD. We allow regulation of which solver to be used for the SMT solver, which provides solutions to formulas with integer predicates. We then use the SMT solver to solve 5 NP-complete problems, including graph coloring, n-queens problem, subset sum, independent set, and the partition problem.

Apart from the internal structure, we also provide a kernel for users to run the SAT solvers directly. We also provide evaluations, tests, a SAT formula parser, and a binary-tree based SAT generator, and documentations to EduSAT in our repository.

The SAT formula parser consists of a lexical analyzer (the lexer) and a syntactic analyzer (the parser) and converts a user-specified SAT formula, following a pre-determined Backus-Naur form (BNF) provided in the kernel, into a list construction of the formula. The SAT generator directly constructs random binary trees (with connectives as non-leaf nodes and atoms as leaf nodes) representing SAT formulas in negation normal form. The user can control the chance each type of connectives occurs, the number of formulas, the number of variables (which can optionally include pure true and false leaf nodes), if repetition is allowed, and the depth of each logic tree generated.

C. Davis-Putnam-Logemann-Loveland Algorithm

Davis-Putnam-Logemann-Loveland Algorithm (DPLL) is the core algorithm for many SMT and SAT solvers. In EduSAT, we only consider the SAT version of DPLL. There are two parts of the search algorithm in DPLL. In the recursive backtracking search, the algorithm first decides an assignment. Then, it deduces the assignment by substituting the atoms in the SAT formula with the assignment. If the valuation returns False, the algorithm backtracks to try another assignment if such new assignment is possible. If the algorithm runs out of assignments, the algorithm returns UNSAT. The heuristics in DPLL revolve around heuristics in deciding and deducing. Our implementation uses a recursive backtracking backbone with a few heuristics introduced to the problem solving progress.

1) Early Termination: One heuristic in assigning is Early termination, which refers to terminating the algorithm if any realization of a variable leads to True or False of a SAT formula regardless the choice of any other variables. If the algorithmn returns False in the early termination, faster backtracking is encouraged. If the algorithm returns True in the early termination, on the other hand, faster path to a solution is found. To give a concrete example, $(A \vee B) \wedge (A \vee \neg C)$ is satisfied by $A =$ True. Thus, the backtracking algorithm can terminate early when A is assigned to True.

To implement this strategy, we eagerly simplify the formula, P, with the heuristics shown in Equation 1, where Q is any Boolean formula, (with commutativity enforced) and carry the simplified formula through the recursive search progress. If at any point the formula is simplified to True, we terminate the algorithm early (unless multiple solutions are needed). If the formula is simplified to False, we encourage early backtracking without further assignments.

$$\begin{gathered} \text{True} \vee Q \rightarrow \text{True} \\ \text{False} \wedge Q \rightarrow \text{False} \end{gathered} \tag{1}$$

2) Unit Clauses: Another heuristic in assigning is Unit Clauses, which states that for any clause left with a single literal, we can simplify the clause by realizing that symbol. More broadly speaking, assignments to variables can lead to simplifications. For instance, in the case of $A =$ False with $(A \vee B) \wedge (A \vee \neg C)$, the clause can be simplified to $(B) \wedge (\neg C)$.

To perform such heuristic, we enhance Equation 2 (with commutativity enforced), where Q is any Boolean formula. in the aforementioned simplification progress.

$$\begin{gathered} \text{False} \vee Q \rightarrow Q \\ \text{True} \wedge Q \rightarrow Q \end{gathered} \tag{2}$$

In both of the aforementioned heuristics, The simplification is applied recursively to encourage simpler formulas in the search progress.

3) Pure literals: One heuristic for deciding assignments is Pure literals, which states that if all occurrences of a symbol in the clause (assuming a negation normal form) have the same sign accross the clause, we can guess that the symbol is True (if the symbol is positive) or False (if the symbol is negative). For instance, in $(A \vee B) \wedge (A \vee \neg C) \wedge (C \vee \neg B)$, the symbol A is pure and positive. By the heuristic, we prioritize the decision of setting A to True over setting A to False in the search algorithm.

To ensure that the pure literals are determined on negation normal forms, we first create an auxiliary binary tree representation of the formula when it is passed to the DPLL solver. The tree is in negation normal form (with negations only appear as the parents of the leaf nodes). Then, we compute find the atoms that are pure positives and pure negatives. When we simplify the formulas, represented in list construction, in the search algorithm, we prioritize to assign True for the pure positive variables. We do not consider prioritization in variable ordering.

D. Reduced Order Binary Decision Diagram

1) Binary Decision Tree: A Binary Decision Tree (BDT) is a tree structure that is commonly used to represent logical formulas. [9] In a BDT, the nodes of the tree represent variables, and the branches represent the values that these variables can take on. Specifically, the branch where its parent node evaluates to true is called the high branch, while the branch where its parent node evaluates to false is called the low branch. The high branch leads to the high child of a parent node, while the low branch leads to the low child of a parent. The leaf nodes of a BDT evaluate to either true or false, which are the outcomes of the test values of each variable.

Our implementation is capable of constructing BDTs for any logic formula expressed using a pre-defined template, in a breadth-first manner. The implementation logic is depicted in Algorithm 1. To construct a BDT, the algorithm requires the logic formula and the variable order as inputs. First, the algorithm creates a root node and a temporary list that stores all intermediate nodes in the tree that do not have left and right children yet. While the list is not empty and the tree has not reached the terminal level, the algorithm dequeues the first item in the temporary list, then adds a low child and a high child to the node. The algorithm continues this process until the temporary list is empty or the tree has reached the terminal level. Finally, the root node is returned as output.

To provide an alternative visualization option, we also offer the functionality to evaluate the constructed BDT and generate a corresponding truth table for the input logic formula. Users can choose to view the truth table if they wish to analyze the logic expression in a tabular form.

Algorithm 1 Binary Decision Tree Construction

```
Parameters: parameter orders $O_p$, logic formula $\varphi$.
BuildOrderedBDT: ($O_p$, $\varphi$)
 1: BDTRoot $\leftarrow O_p[0]$
 2: BDTRoot.left $\leftarrow (O_p[0], 0)$
 3: BDTRoot.right $\leftarrow (O_p[0], 1)$
 4: $RemainingNodes \leftarrow$ [BDTRoot]
 5: while RemainingNodes.length $<$ 2**(number of variables-1) do
 6:     CurrentNode $\leftarrow RemainingNodes.pop(0)$
 7:     CurrentNode.left $\leftarrow$ NewNode($CurrentNode.variable.next, 0$)
 8:     CurrentNode.right $\leftarrow$ NewNode($CurrentNode.variable.next, 1$)
 9:     $RemainingNodes.append$(CurrentNode.left)
10:     $RemainingNodes.append$(CurrentNode.right)
11: return BDTRoot
```

2) Graph Representation of ROBDD: Reduced-Ordered Binary Decision Diagram (ROBDD) is a well-known data structure for representing logical formulas. Unlike BDTs, ROBDDs provide an ordered canonical form of logic representation. More specifically, a logic formula can have different versions of ROBDDs given different parameter ordering. The complexity and memory consumption of a ROBDD depend largely not only on the number of parameters but also on the ordering of parameters. By efficiently ordering the parameters, ROBDDs constructed on-the-run can be significantly more efficient than BDTs, as they require fewer nodes.

A ROBDD is represented with a directed acyclic graph $G = (V, E)$ that consists of multiple decision nodes $\{V_d\}$, two terminal nodes $\{V_t\}$, and the corresponding edges $\{E\}$. The terminal nodes in a ROBDD represent the constants true $\top$ and false $\bot$, the decision nodes represent variables or parameters, and the edges $E \in \{low, high\}$ represent the assignment of truth values to variables, where the low edges represent the assignment of false, and the high edges represent the assignment of true. ROBDDs differ from BDTs in that redundant nodes with identical sub-trees are deleted, and all terminal nodes are combined. In the next section, we will introduce how this is achieved in our framework.

3) Reducing Binary Decision Diagrams: To illustrate the relationship between Reduced Ordered Binary Decision Diagrams (ROBDDs) and Binary Decision Trees (BDTs), as well as to highlight the superiority of ROBDDs over BDTs, EduSAT offers a process for reducing a BDT to a ROBDD.

To transform a Binary Decision Tree (BDT) into a Reduced Ordered Binary Decision Diagram (ROBDD), the process involves a recursive approach with several steps. [9] Firstly, terminal nodes, which represent true or false, are created in the graph. Next, non-terminal nodes are created, and their children are checked to see if they already exist in the graph. All necessary connections are then made simultaneously. Finally, the graph is reduced by removing nodes that have the same left and right children, as well as nodes without any edges. Generally, the elimination rule and isomorphic rule should be followed when constructing a ROBDD. More detailed information can be found in Andersen [8].

E. Satisfiability Modulo Theory Solver

The Satisfiability Modulo Theory (SMT) solver in EduSAT is an extension built upon the aforementioned SAT solvers. In terms of the syntax, the solver only considers integer typed variables with operators discussed in III-A. Also, the SMT solver only provides single solutions in our current framework.

The solver considers each sub-clause with SMT variables as a variable to a SAT problem. For instance, the abstracted SAT problem for $(y0-1>3) \wedge \neg(y1-1<9)$ is $x0 \wedge \neg x1$. The solver first solves the SAT problem using the aforementioned SAT solvers in EduSAT and transforms any negation to the SAT atoms to a positive atom with a negation on the SMT clause. For instance, the aforementioned example, $(y0-1>3) \wedge \neg(y1-1<9)$, is converted by the SMT solver as a conjunction of two constraints, $y0-1>3$ and $y1-1 \geq 9$. Then, the algorithm treats the conjunction of constraints as a Constraint Satisfaction Problem (CSP) and perform an exhaustive search for the valuations of the SMT variables over a user-specified range. In other words, the SMT solver does not guarantee to find a solution if there exists one unless the user-specified search range for the SMT variables is sufficient.

The detailed specification on how the SMT is specified is provided in the github documentations and is not the focus of this report. Please note that even though the specification has an arity of three for each of the SMT clause, the SMT syntax that can be represented is flexible in that temporary variables can be created to form complex SMT clauses (even if such a formation is not the most efficient in terms of time complexity).

F. *Solvers for NP-Complete Problems*

In this section, we discuss the solvers we create for some classical NP-complete problems based on the aforementioned SMT solver. The focus is how we encode such problems to SMT problems.

1) Graph Coloring: Graph coloring is an NP-complete problem. Specifically, we focus on vertex-coloring: Given an undirected graph and a number of colors allowed, assign each node in the graph a color such that there does not exist a pair of adjacent nodes with the same color. We use adjacency lists to represent the graphs. Internally, the solver constructs the SMT encoding in the following way: Firstly, each SMT clause dictates that the chromatic number of a node cannot be equal to that of one of its adjacent node. Secondly, form the SAT representation of the SMT clauses by taking conjunctions of the SMT clauses, which cover all possible edges of the graph. Thirdly, use the SMT solver to solve the encoded SMT problem.

2) N-Queens Problem: In the N-queens problem, given an n by n sized chess board, the algorithm is asked to place n queens on the board such that no two queen attack each other. Inside the solver, each SMT variable represents a column, and the assignmnent to that variable represents the row index of the queen in that column. Then, in the SMT encodings, the solver is supported with representations that no two queen can be in the same row and no two queens can be in the same diagonal. The SAT encoding further abstracts the SMT encodings by connecting the SMT encodings with conjunctions. If the problem is solvable, with the solution, the algorithm transforms the solution into a matrix representation of the chess board.

3) Subset Sum: Subset Sum, another NP-Complete problem, is stated as follows: Given a list, L, of positive integers, find the subset of the list that sum up to a given target value, X. Using the SMT solver, we create a solver for Subset Sum (which is very slow from an application perspective but nonetheless is designed for educational purposes). Please note that this implementation can be inconclusive because of the search space of the SMT solver. Internally, the solver encodes each index of the list to a variable, y_i, with potential values in 0, 1. Then, the SMT encoding denotes the equivalence between $\sum_i(y_i * L_i)$ and the target value, X. The SAT encoding connects the SMT representations via conjunctions.

4) Independent Set: Independent Set is an NP-Complete Problem. We frame our variant of the Independent Set problem: Find a set A, a subset of V, in an undirected graph G = (V, E), where every node in A is not adjacent to any other node in A and the cardinality of A is k. We also frame the maximum independent set problem as the problem to find A with largest possible k. We construct solvers for both the Independent Set problem and the maximum independent set problem.

For the independent set problem solver, the algorithm treats each node to take either values of 0 and 1. The constraints are that the sum of the values of two adjacent nodes is less than 2, and the sum of all the nodes in the graph equals the target cardinality. The SMT representation is then solved using our solver for SMT problems. For the maximum independent set problem solver, the algorithm keeps calling the independent set problem solver with incrementally larger value of target cardinality with a incrementation of 1 until unsatisfiability is detected. Then, the algorithm returns the solution with target cardinality one smaller than the cardinality that triggers the infeasibility.

5) Partition Problem: The Partition Problem is an NP-Complete Problem. We state it as follows without a focus on lists with non-integer constants, since our SMT solver only works on integer predicates: Given a list L, of integers, partition L into two sublists such that the sum of one sublist equals the sum of the other sublist. Like the solver for the Subset Sum Problem, our solver for the Partition Problem is inconclusive due to that the search space of our SMT solver is finite. Internally, the solver encodes each index of the list to a variable (y_i) with a potential values in 0, 1. Then, the SMT encoding represents $[\sum_i(y_i * L[i])] * 2 == \sum_i L[i]$. Thus, All variables that have the same assigned value form one group. The algorithm treats the two groups as the partition of the target list.

IV. Evaluation

In this section we evaluate the performances of the SAT and SMT solvers. We evaluate the accuracy and efficiency of DPLL SAT solver in IV-A, evaluate the accuracy and efficiency of the ROBDD SAT solver in IV-B, and evaluate the accuracy of the SMT Solver (with the DPLL kernel) in IV-C.

A. *Evaluation on the DPLL SAT Solver*

We focus on evaluating both the accuracy and the efficiency of the DPLL SAT solver. In terms of efficiency, we evaluate how the effects of number of formulas, number of variables in each formula, the depth of each formula (as represented by a binary tree representation), and the difference between single vs. multiple solutions affect the speed of the DPLL solver through controlled experiments. In each experimental trial, we generate random Boolean logic trees (with chance of not node, and node and or node held constant at 0.1, 0.45, and 0.45 respectively) and provide solutions to such trees using both a naive tabular solution (equivalently DPLL without heuristic) and the DPLL solver with heuristics and compare the performances. In terms of the accuracy, we evaluate the solutions provided by both the naive solver and the solver with heuristics directly using the evaluation feature of the logic trees. Please note that we only evaluate the accuracy of the formulas that are SAT but assume that the formulas that are UNSAT are solved correctly (although both SAT and UNSAT formulas are taken into considerations when the efficiencies are measured). We will indirectly evaluate the UNSAT formulas through the evaluations of the SMT solver in section IV-C. In experiments with multiple solutions, we also evaluate if the sets of solutions are complete by cross-checking the cardinality of the solution set provided by the naive solver and by the DPLL solver with heuristics.

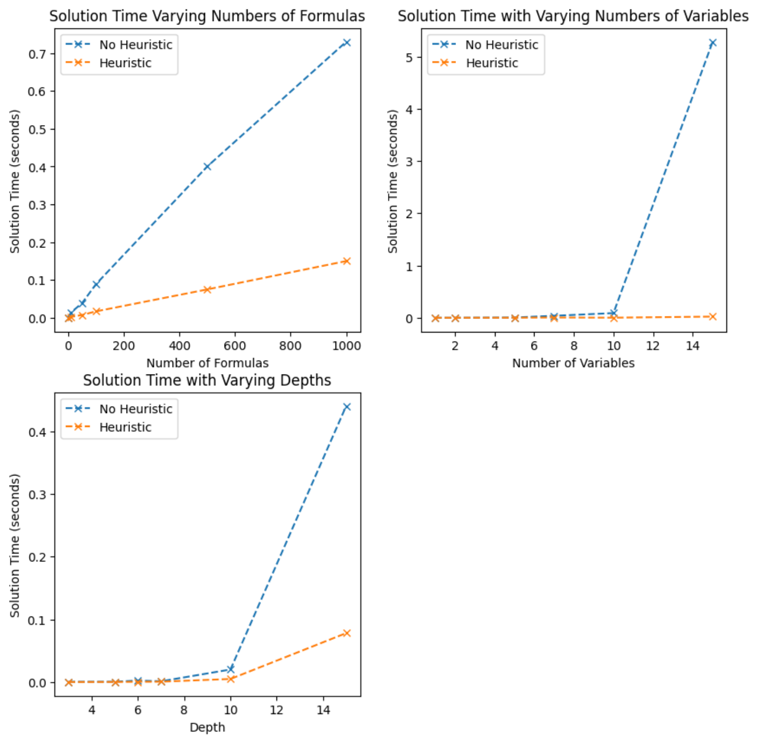

Fig. 2: Efficiency of the DPLL Solver with and without heuristics)

Throughout all trials, the trees are the same for each trail for both the naive solver and the solver with heuristics, but the trees change between the trials. The readers are encouraged to try out the codes in the evaluation folder of the repository of this project in github.

Please note that the evaluation on the DPLL solver was conducted prior to the completion of the ROBDD solver, the introduction of ROBDD as an option to be used in the SMT solver, and the tests. This should have no or minimal effect on the results for the solver submitted to brightspace in terms of the solver performance.

We observe 100% accuracy for both of the naive solver and the solver with heuristics for all trials we perform. Moreover, the solver with heuristics outperform the naive solver in terms of efficiency in all trials we perform.

We showcase the evaluation on the efficiency of the DPLL solver with varying number of formulas solved in Table I. The solvers are required to only provide single answers (given satisfiable problem(s)). As shown in Figure 2, the differences in solving time between the solver without and with heuristics generally increases dramatically with increasing number of formulas solved, suggesting that the a nonlinear improvement enhanced by the heuristics. The increase in solving time for both the nonheuristic and heuristic approach is mostly linear with respect to the increase in the number of formulas solved.

TABLE I: Efficiency of the DPLL Solver with Varying Number of Formulas for Single Solutions

Number of Formulas	Number of Variables	Depth	Solving Time (No Heuristic) in seconds	Solving Time (Heuristic) in seconds
1	5	8	9.89E-05	8.18E-05
10	5	8	0.014	0.0023
50	5	8	0.039	0.0075
100	5	8	0.089	0.017
500	5	8	0.4	0.075
1000	5	8	0.73	0.15

We then demonstrate the evaluation on the DPLL solver with varying number of variables in Table II. Again, the solvers are required to only provide single solutions given satisfiability. As shown in Figure 2, the differences in solving time between the solver without and with heuristics are minimal with increasing number of variables until the number of variables equal 10. This suggest the differences aggregate in a highly nonlinear fashion, and the benefits of the heuristics are highly significant with a large number of variables. The causes of this phenomenon may be the early termination heuristic, which can significantly shorten the amount of time for solving if a solution is found at an early partial assignment. Early backtracking is also encouraged when infeasibility of a partial assignment is found.

We present the evaluation on the DPLL solver with varying tree depth in Table III. The solvers are required to provide only single solutions given satisfiability. As shown in Figure 2, the difference in the solving time between the solvers without and

TABLE II: Efficiency of the DPLL Solver with Varying Number of Variables for Single Solutions

Number of Formulas	Number of Variables	Depth	Solving Time (No Heuristic) in seconds	Solving Time (Heuristic) in seconds
10	1	8	0.00098	0.00095
10	2	8	0.0014	0.00083
10	5	8	0.0059	0.0016
10	7	8	0.038	0.006
10	10	8	0.09	0.0023
10	15	8	5.28	0.023

with heuristics generally increases as the depth of the tree to be solved grows. This phenomenon is especially prominent for tree depth of 10 and of 14. This may be caused by the unit clause heuristic, which may substantially reduce the complexity of a formula with increasing progress in search.

TABLE III: Efficiency of the DPLL Solver with Varying Depths for Single Solutions

Number of Formulas	Number of Variables	Depth	Solving Time (No Heuristic) in seconds	Solving Time (Heuristic) in seconds
10	5	3	0.00042	8.32E-05
10	5	5	0.00051	0.00018
10	5	6	0.0024	0.00046
10	5	7	0.0012	0.00058
10	5	10	0.02	0.0049
10	5	15	0.44	0.078

We now present in Table IV how difference in requiring multiple solutions vs. single solution affects the efficiency of the solver. For all trials requiring multiple solutions, we cross-check if the set of solutions are complete by comparing the cardinality of solution sets solved by the naive solver and by the solver with heuristics. We notice that all solution sets are complete across the trials that require multiple solutions in this evaluation component as indicated by the cross-checks. As demonstrated in Table IV, the solver with heuristics outperform the solver without heuristics significantly in both cases of multiple solutions and single solution required.

TABLE IV: Efficiency of the DPLL Solver with Single Solution vs. Multiple Solutions Required

Number of Formulas	Number of Variables	Depth	Multiple or Single	Solving Time (No Heuristic) in seconds	Solving Time (Heuristic) in seconds
10	5	8	multiple	0.021	0.0035
10	5	8	single	0.0091	0.0019
3	7	10	multiple	0.083	0.014
3	7	10	single	0.021	0.0012
1000	3	5	multiple	0.073	0.039
1000	3	5	single	0.041	0.019

B. Evaluation on the ROBDD SAT Solver

This section provides a detailed description of the demonstrations and evaluations of the ROBDD functionalities in EduSAT. All experiments were performed on an Intel Core i9-10850K CPU running at 3.60GHz. The python `networkx` package was used to represent and visualize the ROBDD graph. The accuracy and efficiency of the ROBDD SAT solver were evaluated using the same format as DPLL. To measure efficiency, we recorded the runtime from the conversion of a BDT to a ROBDD to the completion of finding all SAT solutions. To evaluate accuracy, we generated logic formulas with varying numbers of parameters and depths, then verified the solutions provided by the ROBDD solver by substituting them back into the formulas. Here, the depth of a logic formula is defined as the number of nested connections (with $\wedge$ or $\vee$).

We present visual demonstrations of the ROBDD functionality in Fig.3-Fig.6. Specifically, Fig.3 and Fig.4 represent the logic formula $f(x_0, x_1, x_2) = (\neg x_0 \wedge \neg x_1 \wedge \neg x_2) \vee (x_0 \wedge x_1) \vee (x_1 \wedge x_2)$, while Fig.5 and Fig.6 showcase a more complex formula, $f(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7) = ((x_0 \wedge x_1) \vee (x_2 \wedge x_3) \vee (x_4 \wedge x_5) \vee (x_6 \wedge x_7))$. Although our tool offers truth table visualizations, we find that truth tables become significantly less efficient as the number of parameters increases. Overall, the visualization of ROBDDs provides a clear and concise understanding of the logic formula structure, making it easier for users to identify potential issues with their logic.

Table. V presents the runtime of the ROBDD solver for finding single and multiple solutions. For the multiple solutions task, we find all possible solutions to a logic formula, while for the single solution task, we always provide the *shortest and simplest* solution. As a result, we observe a slight increase in runtime for the single solution task compared to the multiple solution task. It is also evident that the total solving time generally increases as the number of formulas increases, which is expected. We note that the ROBDD solver achieved 100% accuracy in both the single solution task and the multiple solution task.

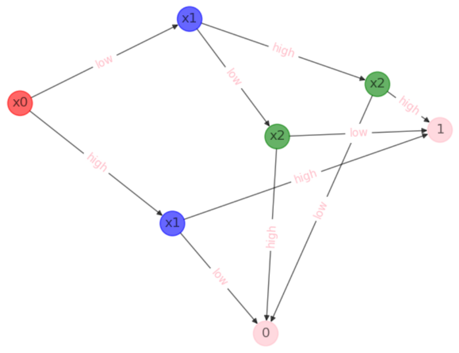

Fig. 3: ROBDD with variables x_0, x_1, x_2.

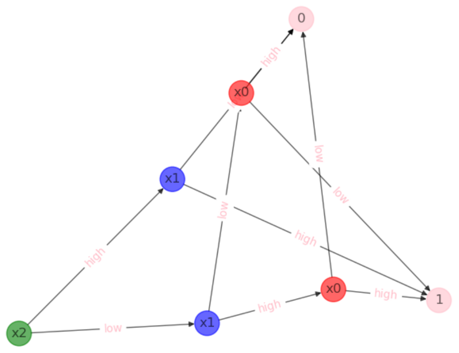

Fig. 4: ROBDD with variables x_2, x_1, x_0.

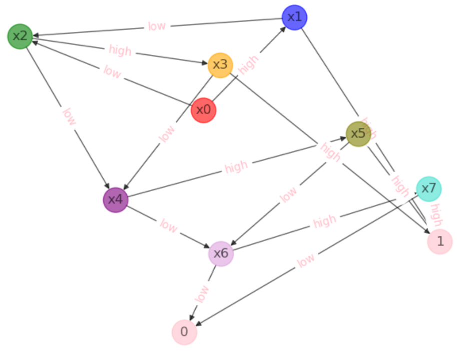

Fig. 5: ROBDD with variables $x_0, \ldots x_7$.

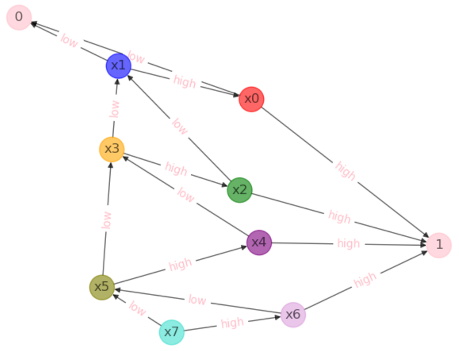

Fig. 6: ROBDD with variables $x_7, \ldots x_0$.

TABLE V: ROBDD solver runtime with varying number of formulas for single and multiple solutions.

Formulas	Parameters	Depth	Single Solution (s)	Multiple Solutions (s)	Accuracy (Single)	Accuracy (Multiple)
1	3	3	0.00017	0.00011	100%	100%
10	3	3	0.00098	0.00071	100%	100%
50	3	3	0.00458	0.00435	100%	100%
100	3	3	0.00815	0.00987	100%	100%
500	3	3	0.04514	0.04410	100%	100%
1000	3	3	0.08647	0.08902	100%	100%

Table. VI presents the runtime of the ROBDD solver for varying formula lengths and number of parameters. We report results for both single and multiple solutions for both cases. In the first three rows, we fix the number of formulas and the number of parameters, and change the depth of the formulas. We observe a slight increase in runtime for finding multiple solutions. However, the change is *not significant*. This can be attributed to the fact that the complexity of a formula does not always increase as the depth increases. For example, a formula that repeats the same logical operation at every level (e.g., $(\varphi_1 \wedge (\varphi_2 \wedge (\varphi_3 \wedge \ldots))))$ may not become significantly more complex as the depth increases. In rows 5-7, we fix the number of logic formulas and change the complexity of the formulas by increasing the number of parameters and depth. We notice that the runtime for both single and multiple solutions increases as these two hyperparameters increase, which is as expected. We performed a cross-validation study for all experiments reported above and found that our results achieved *100%* accuracy.

TABLE VI: ROBDD solver runtime with varying formulas lengths and number of parameters.

Formulas	Parameters	Depth	Single Solution (s)	Multiple Solutions (s)	Accuracy (Single)	Accuracy (Multiple)
10	5	5	0.00255	0.00249	100%	100%
10	5	6	0.00266	0.00271	100%	100%
10	5	7	0.00243	0.00276	100%	100%
10	2	1	0.00056	0.00052	100%	100%
10	3	3	0.00096	0.00082	100%	100%
10	7	6	0.00771	0.00828	100%	100%

C. Evaluation on the SMT Solver

We evaluate the accuracy of the SMT solver through 8 case studies with results listed in Table VII, where UNSAT indicates that the given formula is determined to be unsatisfiable by the SMT solver. We use the DPLL SAT solver as the kernel throughout this evaluation component. Since the SMT solver only returns single solutions, we evaluate the solutions found by the solver directly against the specified SMT requirements. The upper and lower bounds in Table VII denote the search range for the SMT variables (which need to be adjusted accordingly to avoid false results while ensuring that reasonable solving time). The solutions provided by the solver can be nondeterministic. Please note that we repeat a substantial number of test case formulas in our test for SMT Solver in EduSAT.

Please note that the evaluation on the SMT solver was conducted prior to the completion of the ROBDD solver, the introduction of ROBDD as an option to be used in the SMT solver, and the tests. This should have no or minimal effect on the results for the solver submitted to brightspace in terms of the solver performance.

TABLE VII: Case Studies for SMT Solver Evaluation

Equation	Lower Bound	Upper Bound	Solution Provided by the Solver	Solver Correctness
$(y_1 - 2 = y_2) \wedge (y_2 + y_1 > 5)$	0	10	$y_1 : 4, y_2 : 2$	Correct
$(y1 - y2 \neq y1 + y2) \wedge (y2 + y1 = y1)$	0	10	UNSAT	Correct
$(\neg(y_1 < 6) \wedge (y1 - y2 \geq 10)) \wedge (3 < y2)$	-10	20	$y1 : 14, y2 : 4$	Correct
$(y1 \neq 10) \wedge (y1 * y2 = 100)$	0	30	$y1 : 4, y2 : 25$	Correct
$(y1 = 5) \wedge (y2 // y1 = 10)$	-100	100	$y1 : 5, y2 : 50$	Correct
$(y1 <= 5) \vee (y2 <= 3)$	0	10	$y1 : 0, y2 : 4$	Correct
$(y1 + y2 = y4) \wedge (y4 + y3 = 10)$	0	10	$y1 : 0, y2 : 0, y3 : 10$	Correct
$(y1 = y2) \wedge (\neg(y1 = y2))$	0	10	UNSAT	Correct

As shown in Table VII, all solutions to the test cases satisfy the requirements provided in the format of the equations.

V. Limitation

a) Limitation of the DPLL SAT Solver, the SMT Solver, and the solvers for NP-complete problems: There are many advanced heuristics, which we do not implement in the current framework of EduSAT, that can be applied to the implementation of a SAT solver. For instance, we do not explicitly implement the Boolean Resolution (though a reduction in the simplified process may be possible). To accelerate the DPLL SAT solver, we can perform smart variable ordering in the search progress and implement caching if we continue this work in the future. In terms of the SMT solver, we only allow integer predicates with a strict syntax. Moreover, in our current implementation, it is not possible to specify a long arithmetic formula without temporary variables. If we will continue to work on EduSAT in the future, we can expand the syntax of the SMT clauses acceptable by the solver and implement an SMT solver with a higher efficiency by using DPLL(T). In the current implementation, we only provide solver abstractions for 5 NP-complete problems apart from the SAT and SMT problems. In the future, if we plan to improve EduSAT, we can use the SMT solver to solve more NP-complete problems.

b) Limitation of the ROBDD SAT Solver: In the EduSAT tool, we demonstrate the process of reducing a BDT by building its corresponding ROBDD. However, constructing ROBDDs directly is often more efficient [9]. As the next step, we plan to implement the feature to construct ROBDDs on-the-run and compare their runtime with the existing approach to further demonstrate the advantage of building ROBDDs directly. This will provide users with a more efficient option for solving NP-complete problems, especially for more complex ones. Additionally, to enable users to better compare BDTs with their equivalent ROBDDs, we aim to provide visualizations of BDTs as binary trees in the future. This will allow users to directly compare the structure and size of the two representations and gain a better understanding of how ROBDDs are more compact and more efficient than BDTs. Overall, these enhancements will make the EduSAT tool more effective and user-friendly.

VI. Conclusion

In conclusion, in this project, we implement EduSAT, which is created for educational purposes. We introduce a DPLL SAT solver, a ROBDD SAT Solver, an SMT solver on integer predicates, and solver abstractions for 5 NP-complete problems other than SAT and SMT. We provide comprehensive testings, documentations, and evaluations. In the evaluations, we present the efficiency and accuracy for both the DPLL and the ROBDD SAT Solver. We also evaluate the accuracy of the SMT solver through 8 case studies. Through the evaluations, we also analyze how different properties of formulas contribute to the efficiency of the SAT solvers. All SAT and SMT solvers show 100% accuracy in the evaluations.

VII. Acknowledgement

Throughout the paper, we use the lecture materials from Vanderbilt University CS 6315 provided by Professor Taylor Johnson and our own documentation from the GitHub repository for this project. For more acknowledgements (not directly related to this report), please read the codes in the repository.

References

[1] J. Jain, A. Narayan, M. Fujita, and A. Sangiovanni-Vincentelli, “Formal verification of Combinational Circuits,” Proceedings Tenth International Conference on VLSI Design.

[2] M. Kamali, L. A. Dennis, O. McAree, M. Fisher, and S. M. Veres, “Formal verification of autonomous vehicle platooning,” Science of Computer Programming, vol. 148, pp. 88–106, 2017.

[3] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “seL4: formal verification of an OS kernel,” Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009.

[4] L. de Moura and N. Bjørner, “Z3: An efficient smt solver,” Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340, 2008.

[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, “CVC4,” Computer Aided Verification, pp. 171–177, Jul. 2011.

[6] B. Dutertre, “Yices 2.2,” Computer Aided Verification, pp. 737–744, 2014.

[7] A. Höfler, “SMT Solver Comparison,” Jul. 2014.

[8] Andersen, Henrik Reif. ”An introduction to binary decision diagrams.” Lecture notes, available online, IT University of Copenhagen (1997): 5.

[9] Alur, Rajeev. Principles of cyber-physical systems. MIT press, 2015.