Implementation and analysis of a SAT solver

Santiago García (s4450361) and Lucas Idsinga (s4626273)

Leiden University

Abstract. In this final project of the Software Verification course, we implement our own SAT solver. We create a DPLL-based algorithm with unit propagation, pure literals, and the 2-Watched Literals strategy. We also implement different decision heuristics when choosing the next literal, putting emphasis on doing it with a positive memory usage.
The report shall analyse its performance depending on the speed and the memory. To be able to work efficiently and control the memory, the SAT solver will be implemented in C, as it has a more predictable and efficient memory usage than other programming languages.

Keywords: SAT · Literal · Clause · DPLL · Unit Propagation · Pure Literals · 2-Watched Literals · Stack.

1 Introduction

The Boolean Satisfiability Problem (SAT) is a cornerstone problem in Computer Science and Mathematical Logic, playing a central role in fields such as Artificial Intelligence, Verification, and Optimisation. The SAT problem consists of being given a Boolean formula, determining whether there exists an assignment of variables that makes it true. Its origins go back to the 1800s, with the formalisation of Logic by Boole [3]. During the 20th century, several important algorithms to solve it were proposed, with the most notable ones being the DP (Davis, Putnam) algorithm in 1960 and the DPLL (Davis, Putnam, Logemann, Loveland) algorithm two years later. At the end of the century, Silva and Sakallah added a clause learning method to the previous algorithm, improving its performance. This method receives the name of Conflict-Driven Clause Learning (CDCL). However, the main result from this theorem was obtained in 1971, when Stephen Cook proved in [1] that the SAT problem is an NP-complete problem.

A SAT solver is a tool designed to determine if there exists an assignment of variables that makes a Boolean formula true. This report presents the implementation of a SAT solver, outlining the underlying algorithms, data structures, and design decisions involved. Through this project, we aim to explore the key techniques of our DPLL-based algorithm, such as unit propagation, pure literals, backtracking, and 2-Watched Literals. We implement the algorithm in C, which will allow us to deal in a more efficient way with the memory usage problem. We shall explain the different approaches that we had, showing the different results

obtained. To conclude, we will discuss potential improvements and limitations of the current implementation.

2 Proposed Algorithm

In this section, we will briefly explain our DPLL-based algorithm from a theoretical perspective, following [3]. We will not extend in this part, as later we will go deeper into every algorithm, explaining the code.

2.1 CNF Formulas

Normally, SAT solvers operate with Boolean formulas in a Conjunctive Normal Form. The formula is a conjunction of clauses, and each clause is a disjunction of variables. The DPLL algorithm determines whether there exists an assignment of truth values to variables such that the entire formula evaluates to true. Our algorithm works with the DIMACS format, which is a small modification of it, as it is a CNF representation of the formula, where each line represents a new clause and the first line has the type p cnf <variables> <clauses>. Working with them, we are not losing generality, as the Tseitin transformation [6] can make an equivalent, polynomial-sized CNF for every Boolean formula.

2.2 DPLL algorithm

The DPLL algorithm, created in 1962 by Davis, Putnam, Logemann, and Loveland, is a recursive backtracking method used to solve the SAT problem. It forms the foundation of many modern SAT solvers. It solves the memory consumption that the DP algorithm had, due to the duplication of the clause size in every resolution step. It introduces the unit propagation and the pure literal rules. If there is no pure literal, it chooses heuristically a variable to assign and propagate. If it gets a contradiction, it backtracks and tries with the negation of the value received by the first variable. If it also fails, it backtracks again, or it returns UNSAT if there is no backtrack possible. The algorithm terminates with SAT if the formula does not yet have any clauses to satisfy. These techniques significantly improve practical performance when searching for a satisfying assignment compared to DP.

2.3 Unit Propagation

The unit propagation rule analyses if there is any clause that is formed only by a single literal. We will label that clause as a unit clause. If we want to satisfy a formula, every literal contained in a unit clause must satisfy it. By resolving unit literals early, it minimises recursion depth and enhances solver performance, ultimately improving search efficiency. The algorithm assigns and propagates the assignment throughout the formula. This step may trigger a cascade of further unit propagations.

2.4 2-Watched Literals

The 2-Watched Literals is an optimisation technique used by modern SAT solvers to handle the Boolean constraint propagation efficiently, the bottleneck of any SAT solver. This strategy helps reduce the computational cost of maintaining and updating clauses when variables are assigned to values.

Each clause, if possible, has two, and only two, watched literals at any given time. If there are not two watched variables, then the clause will be empty or a unit clause. When a variable is assigned a value that affects one of the watched literals, the solver checks if the clause is still satisfied or if it needs to look for another literal. If it cannot, we apply unit propagation over that literal; if we encounter some empty clause, we return UNSAT. This prevents unnecessary scanning of all literals in every clause every time we assign a variable. It avoids unnecessary backtracking and improves performance, as it should resolve big SAT formulas faster.

2.5 Pure literals

A literal is pure if it appears in the formula only as positive or as negative. Assigning such literals to true (if positive) or false (if negative) simplifies the formula without getting any contradiction. As in the previous function, after fixing a pure literal, there exists the possibility that new literals appear, so we have to redo the analysis.

3 Implementation

We shall explain in detail the implementation of the algorithm in C, dividing into subsections each part. The pseudocode idea was from [3] and the lecture slides.

3.1 Data Structures

First, we define the data structures that we will use throughout the whole project.

- The literals are represented as an integer that represents their number and a boolean that states if it appears as a negation or not.
- In the clause struct, we store their size, and we use a pointer to a list of literals that will be contained in that clause. We also add a boolean that acts as a flag that stores whether the clause is already satisfied or not.
- The formula stores the complete Boolean formula as a conjunction of clauses. It tracks the number of variables (numVars) and clauses (numClauses).
- We define a WatchTable struct, which contains a table which maps a literal (separated with affirmed and negated variables), to a list of clause indexes that are watching those literals. This is done to minimise the memory required by the table.

– We define an undo stack in order to obtain some memory optimisations, which also leads to time improvements. It is composed of three structures. The first one, UndoType, is an enum that defines reversible actions (assignment, clause satisfaction, etc), the second, UndoEntry, is a struct that represents a single undo operation, while the last, UndoStack, is a pointer to a list for backtracking operations. The stack, of type UndoStack, is used to record variable assignments, so that they can be undone later if a conflict is encountered during the search.

3.2 Parse Function

The parse function is designed to read and parse a formatted CNF formula in the DIMACS format from a file. It constructs a Formula data structure by extracting variables and clauses, ensuring efficient storage and management. It tries to open the given file, returning NULL if it cannot be accessed. It reads lines until finding the line p cnf numVars numClauses. It extracts and stores from it the values of numVars and numClauses, characteristics of our formula structure. We iterate through the lines containing literals, ignoring empty lines, as we do not want to read them as an empty clause, comments and headers. We also fix an important edge case at the end of the parser. If we do not find as many clauses as expected, we adjust the numClauses variable so that the solver does not analyse these cases as if we have empty clauses, which would result in always returning UNSAT. Before returning the formula, the file is closed.

3.3 Eliminate duplicate/superset clauses

When dealing with formulas in the CNF format, one possible source of redundancy is the presence of duplicate clauses, and, most commonly, clauses that are supersets of others [2]. In the implementation, after the parse function, we pre-process the formula to eliminate these redundancies. The search space is reduced, leading to decreased memory usage and faster resolution. This optimisation could be especially beneficial if we are dealing with a large formula, as the possibility of having these clauses that are super sets of others is high, and can have a significant cumulative impact.

3.4 Unit propagation

The unit propagation function, which should propagate unit clauses to simplify its structure, receives as inputs the formula, the array tracking the variable assignments, where we assign the value -1 if it is unassigned, and the undo stack to facilitate the backtracking. It returns false in case that a conflict occurs during unit propagation (an empty clause unsatisfied appears) or true if the propagation succeeds.

The implementation follows the next steps:

1. The function begins in a loop until no further progress is made, and we make progress if we find a unit clause propagating that assignment, resetting the cycle every time we do progress.
2. We process every clause, storing the number of unassigned variables, the last unassigned variable, useful in the case that we are in a unit clause to do backpropagation more efficiently, and if the clause is satisfied or not.
3. If the clause is satisfied, we skip that clause. On the other hand if it is not satisfied and we did not find any unassigned variable, we return UNSAT, as we found a conflict during unit propagation.
4. If the studied clause was a unit clause, the value of unassigned variables is 1, we store the assignment of that variable to our stack and mark progress as true.

We should call the unit propagation function at the beginning of the DPLL function. Next, we shall explain the 2-Watched Literals function, which is an optimisation of this method that can substitute it, comparing in the result section the difference in time and memory performance for both methods.

3.5 2-Watched Literals

The 2-Watched literals method, as the unit propagation, receives as inputs the formula, the array tracking the variable assignments, and the undo stack to facilitate the backtracking. It returns false in case that a conflict occurs during unit propagation (an empty clause unsatisfied appears) or true if the propagation succeeds.

The implementation follows the next steps:

1. We start creating a dynamic queue to store the variables that form unit clauses and need to be propagated further.
2. The function loops over all the clauses in the formula. If the clause is satisfied, it continues. If not, it examines its literals, searching for unassigned variables, copying all the time the value of the last variable. To be able to push it into the queue if that is the only unassigned value, so we are in a unit clause.
3. We loop over the propagation queue until it is not empty:
   (a) We assign its value depending on its sign; non-negated literals are assigned to true, while negated literals are assigned to false.
   (b) We update the list of watched literals, looking for the clauses that are watching the opposite of the assigned literal. The function locates the corresponding index of it in the watch table and gets the list of clauses that are watching it.
   (c) We loop throughout the literals of that clause, looking for new literals to watch. If we were unable to find any alternative, then either the clause is satisfied, so we can skip it, or all the variables are assigned in a way that makes the clause false, so we are in a contradiction, returning false. Also, if we cannot watch a new literal, but the other literal is not assigned yet, then we detect a new unit clause, increasing the propagation queue.

   (d) We try to replace the current watched literal by searching in the clause for either a literal that already satisfies the clause, or an unassigned variable, taking the first variable that meets some of the previous conditions. During this process, the clause is removed from the current watch list and added to the new one.
   (e) If no new literal can be found, then it is a unit clause, so the variable is immediately assigned and put in the queue. Otherwise, if the other literal is already unassigned in a way that leads to a conflict, the propagation terminates with a failure.
4. Once the propagation queue is empty, we have successfully propagated every unit clause without detecting a conflict. As in the unit propagation function, we return true after freeing the memory of the queue.

3.6 Pure Literals

Like the previous two methods, its inputs are the formula, the list of variable assignments, and the undo stack. It always returns true, we cannot find any contradiction in this method, only simplifies trivially the final formula.

The implementation follows the next steps:

- We create two auxiliary arrays, positive_units and negative_units, that store whether a variable appears as a positive literal or as a negative literal in the formula. Only looping through the unsatisfied clauses.
- With the previous auxiliary lists, we create the list of booleans lit_purity that stores if the variables appear only with one polarity, skipping already assigned variables. If it only appears in the negative list, we give that variable the assignment false, pushing it into the stack. Conversely, if it only appears as positive, we do the same with true.
- We update the clauses that contain that pure literal, as now they are satisfied, pushing that to the stack.
- To conclude, we liberate the memory used for positive_units, negative_units, and lit_purity, and return true. As no contradiction will appear in this function.

3.7 DPLL

The DPLL follows the pseudocode of the lectures, Fig. 1, with a small optimisation. The second if, which analyses if there is some empty clause in our formula, can be removed. As we have already checked that in our unit propagation or 2-Watch Literals functions.

The inputs of our function will be the formula, the array that contains the assignments, the undo stack, and, in the case where the implemented unit propagation function is the 2-Watched Literals, the function also receives the list of watched literals. In order to work with it for the experiments section, the function returns an enum, with SAT, UNSAT, and TIMEOUT as options, where we establish the timeout in one hour.

Algorithm 3 DPLL(CNF $\varphi = \{c_1, \ldots, c_m\}$)

```
Unit propagation
Fix pure literals
if φ = ∅ then
  | return SAT
if {} ∈ φ then
  | return UNSAT
Select x_i ∈ {x_1, ..., x_n}
Select v ∈ {0, 1}
if DPLL(φ[x_i := v]) = SAT then
  | return SAT
else
  | return DPLL(φ[x_i := 1 − v])
```

Fig. 1. Pseudocode of the DPLL algorithm.

The most important part of this function is the heuristic used to select the next variable and its value. It is a critical decision, as cleverly choosing the next literal can imply a greatly sped up. However, this problem is as hard as SAT itself. We implemented two different heuristics. The first one takes the first literal unassigned, considering the classical numerical order, and gives it the false value first, backtracking later if necessary. The second one, more complex, uses the idea of the Variable State Independent Decaying Sum (VSIDS). We create a sorted list of the literals, depending on their appearances in the formula, before calling the DPLL function. Then, when the function returns the literal with the greater number of appearances that is still unassigned. The most important difference between VSIDS and this method is that VSIDS do not study the whole formula at the beginning, analysing it clause by clause, giving special importance to recently added clauses. The difference is almost null in practical terms.

3.8 Memory optimisations

At the beginning of the project, we realise that to be able to backtrack, we need to save in some way how our formula was before the assignment of some variable, to be able to recover its value and try with its negation.

The first primitive idea was to clone the formula each time, saving all the time we assign some value a literal as the formula was before that decision. It backtracked, however, the memory consumption was huge, as we see in Fig. 2, which is a formula that was SAT, and in Fig. 3, which is an UNSAT formula. It is important to highlight that the fluctuations of memory usage that the figures display are due to the backtrack of the DPLL algorithm.

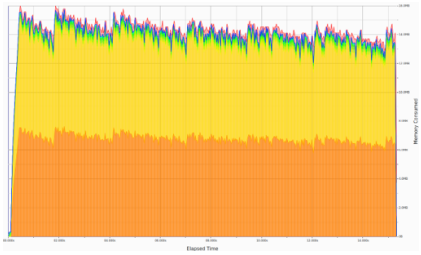

Fig. 2. Plot that shows the memory and time used to solve the 16-Queens problem using the clone idea.

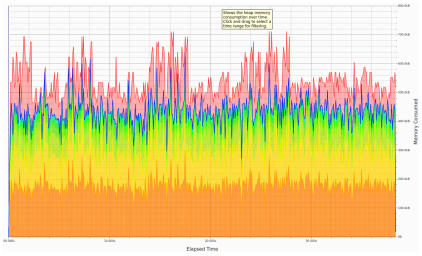

Fig. 3. Plot that shows the memory and time used to solve a UNSAT formula using the clone idea.

Trying to solve these problems, we came up with the idea of storing the actions taken by the algorithm in a stack. By storing a pointer to a checkpoint in a stack, we can pop actions, using the data stored within to undo that action, until we reach the checkpoint. By doing this, we avoid having to store unnecessary data such as with cloning the entire formula. We see the results obtained by applying the same SAT problems as before, but now using the stack.

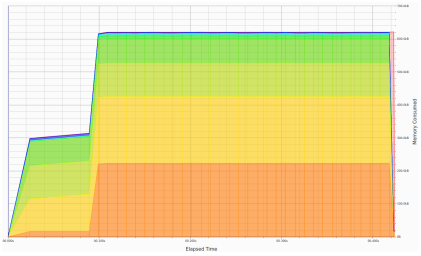

Fig. 4. Plot that shows the memory and time used to solve the 16-Queens problem using the stack idea.

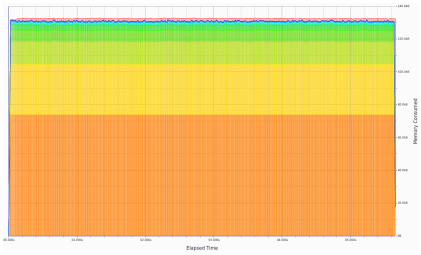

Fig. 5. Plot that shows the memory and time used to solve a UNSAT formula using the stack idea.

Not only did the algorithm improve the memory consumption, from 16 MB using the clone idea to 600 kB with the stack in the 16-Queens problem, or 720 kB compared to 120 kB in their peak in the UNSAT problem, but it also improved the speed of solving the problem. As for the clone algorithm, the SAT and UNSAT problems took 15.2 and 40 seconds, while the stack idea took 0.42 and 6 seconds, respectively. The performance increase is likely due to no longer having to create a deep copy of the formula, which can be computationally expensive.

Moreover, the majority of the memory consumed in the stack version, the orange part of the plots, is due to storing the formula. So there is not much room

for improvement in terms of memory, besides some small optimisations in our code.

4 Experiments

This section describes the experiment setup, alongside the results of the experiments.

4.1 Experiment Setup

The program is run on test cases obtained from the SATLIB - Benchmark Problems dataset hosted by The University of British Columbia [4]. The chosen test cases consist of 4 sets of problems are chosen from the *Uniform Random-3-SAT* data set, as this set provides both satisfiable and unsatisfiable problems of the same format. For both the SAT and UNSAT cases, the first 10 problems in each of the following sets are chosen and used in the experiment:

- 50 variables, 218 clauses
- 100 variables, 430 clauses
- 175 variables, 753 clauses
- 250 variables, 1065 clauses

A cut-off timer of 1 hour (3600 seconds) is chosen. If the execution time exceeds the cut-off timer, the program terminates early. The mean score excluding the cut-off values is returned to show the performance of the examples that do terminate.

The experiment is run according to the following set up:

- **OS**: Windows 11 Home with WSL 2.7.5.0 running Ubuntu 24.04.2 LTS
- **CPU**: Intel i9-13900K run on a single thread
- **RAM**: 2x32GB DDR5 5600 MHz
- **Solver**: DPLL-based solver implemented with a 2-Watched Literals algorithm
- **Compiler**: gcc (Ubuntu 13.3.0-6ubuntu2∼24.04) 13.3.0

4.2 Results

The experiment is then performed according to the setup described in Subsection 4.1, resulting in the values shown in Table 1. In this table, the problem sets that triggered timeouts are shown. Timeouts are not calculated into the mean, so the mean represents the average time of successful completions.

Tests were performed with both heuristics described in Subsection 1, however, early on in experimentation it was found that the heuristic which looks for the First Variable Unassigned would timeout on simple cases. As such, the remainder of the experiments were only run using the Most Appearance heuristic.

Table 1. Mean results of the SAT solver across 10 variations of problems of each test case. In the result of a timeout, the result is not computed into the mean time.

Test Case			Timeouts	Mean Time
Variables	Clauses	SAT/UNSAT	(1 hour)	(seconds)
50	218	SAT	0	7.646e-3
50	218	UNSAT	0	9.123e-3
100	430	SAT	0	8.170e-2
100	430	UNSAT	0	1.737e-1
175	753	SAT	0	6.276e0
175	753	UNSAT	0	3.463e1
250	1065	SAT	0	1.164e3
250	1065	UNSAT	3	2.114e3

These results can be compiled into a plot as shown in Figure 6, where the error bars are used to represent the range of results, where any experiments that resulted in a timeout were set to 3600 seconds.

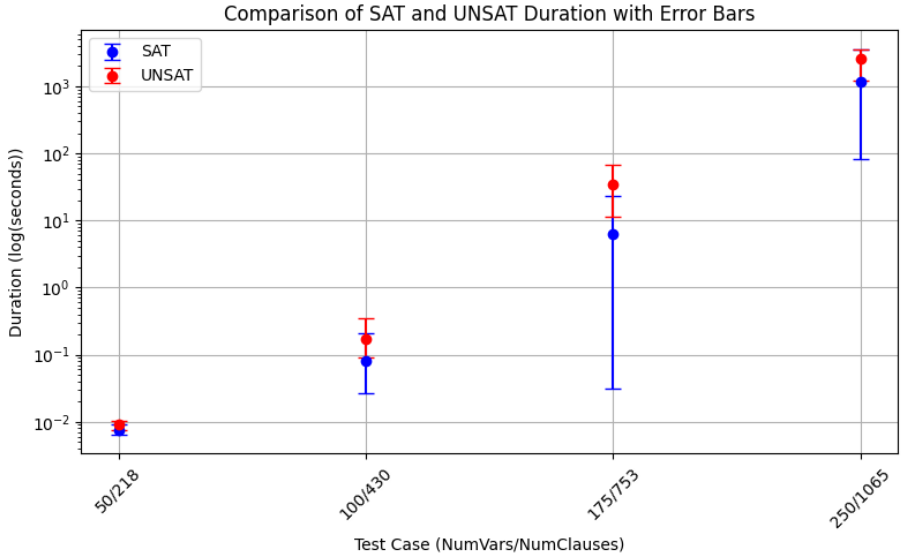

Fig. 6. Plot representing the average performance of the SAT solver across the test cases for both SAT and UNSAT results. Error bars represent the range of results, with a cut-off of 1 hour. Any results that triggered the cut-off are set to 1 hour.

5 Conclusion and Discussion

The goal of the project was to create a SAT solver based on the DPLL algorithm. The SAT solver was designed to be able to take a CNF file in the DIMACS format as input, and compute the satisfiability of the problem, tracking the time spent to do so. Using the 2-Watched Literal algorithm, the SAT solver is able to quickly find satisfying assignments to unit clauses. For backtracking, a memory efficient undo stack is implemented, which tracks only the necessary information to be able to revert the formula in memory to a previous point in time.

The SAT solver was then tested on a total of 80 satisfiable and unsatisfiable problems across 4 conditions. With a cut-off timer of 1 hour, only 3 of the 80 results were not able to be computed within the time limit. These 3 timeout examples all occurred on the largest unsatisfiable test cases. Based on the results shown in Table 1, this test case is the one that takes the longest on average, even when discounting these results (which implies a lower average time). As such, it can be expected that if the cut-off timer was longer, that the solver should be able to find a solution to these problems.

From these results, it can also be observed that solving SAT problems tend to be faster than solving UNSAT problems. In the case of the implementation of the SAT solver, this is likely due to the fact that for returning a SAT result it is only required to propagate a single set of satisfying assignments. For the UNSAT case, it is required to show that there is a set of clauses that are mutually exclusive, which is harder to show without clause learning.

The results seen in Figure 6 also show an increase in the variance of the results. One potential explanation for these results is that the problem grows in size, the difference in the time it takes to solve a simple problem and a complex problem grows as well, as there are more clauses and variables to iterate through. Another explanation could be the choice of the heuristic that DPLL uses to decide which variable to iterate. With larger problems, it could be observed that the choice of heuristic matters more than it does for smaller problems.

6 Future Work

While the current DPLL-based solver demonstrates promising performance with its implementation of unit propagation using 2-Watched Literals, pure elimination, and clause reduction, several key enhancements could be explored to reach a better SAT solver. The most important are for future work is the implementation of a Conflict Driven Clause Learning, CDCL. It would enable the solver to analyse conflicts, learn new clauses that prevent the same conflicts from recurring. It can greatly improve the searching process, especially on complex or large instances. For UNSAT instances, possibly the difference would be huge. We attempted to do it, however, we could not integrate the clause learning strategy,

failing in our attempt, and implementing 2-Watched Literals instead. Some useful references on this problem were [3] and [5].

On the other hand, the most appearances strategy, which follows the same idea of the VSIDS heuristic, gets some promising results over the "trivial" heuristic. However, we could also try to implement more heuristics to select the next variable to test the differences in its performance. One different approach to tackle the selection problem could be applying some Reinforcement Learning from Algorithm Feedback heuristics [7]. Some small optimisations on the memory or the parser could create a better SAT solver, but probably it will not make a huge difference, as the current SAT solver deals positively with the memory usage problem.

References

1. Cook, S.A.: The complexity of theorem-proving procedures. In: Harrison, M.A., Banerji, R.B., Ullman, J.D. (eds.) Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA. pp. 151–158. ACM (1971). https://doi.org/10.1145/800157.805047, https://doi.org/10.1145/800157.805047
2. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3569, pp. 61–75. Springer (2005). https://doi.org/10.1007/11499107_5, https://doi.org/10.1007/11499107_5
3. Franco, J., Martin, J.: A history of satisfiability. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 3–74. IOS Press (2021). https://doi.org/10.3233/FAIA200984, https://doi.org/10.3233/FAIA200984
4. Hoos, H., Stützle, T.: SATLIB: An online resource for research on SAT, pp. 283–292 (04 2000)
5. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001. pp. 530–535. ACM (2001). https://doi.org/10.1145/378239.379017, https://doi.org/10.1145/378239.379017
6. Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg (1983). https://doi.org/10.1007/978-3-642-81955-1_28, https://doi.org/10.1007/978-3-642-81955-1$_2$8
7. Tönshoff, J., Grohe, M.: Learning from algorithm feedback: One-shot sat solver guidance with gnns (2025), https://arxiv.org/abs/2505.16053