CDCL implementation in Python

Vinicius Silva Gomes

Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
vinicius.gomes@dcc.ufmg.br

1 Introduction

Given a finite set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$ and a propositional formula φ expressed in Conjunctive Normal Form (CNF), composed of a set of clauses $C = \{c_1, c_2, \ldots, c_m\}$, where each clause c_i is a disjunction of literals (variables or their negations), the satisfiability (SAT) problem can be defined as determining whether there exists an assignment $\alpha : X \to \{0, 1\}$ such that φ evaluates to true, i.e., $\varphi(\alpha) = \top$. The problem is satisfiable (SAT) if an assignment α like that exists and is unsatisfiable (UNSAT) if not. It was the first NP-complete problem to be proven [1]; nevertheless, it remains a fundamental problem in various fields, such as formal verification of software and hardware, optimization, artificial intelligence, and circuit design.

The SAT problem can be tackled using various methods, which have evolved to address its inherent complexity. Early approaches like pure backtracking exhaustively explored all variable assignments but were inefficient for large instances due to combinatorial growth. The DPLL algorithm [2] improved this by introducing unit propagation to simplify formulas by resolving unit clauses. However, it still faced limitations by revisiting conflicting states repeatedly.

To overcome these challenges, CDCL (Conflict-Driven Clause Learning) [3] was developed, building on DPLL with innovations like conflict-driven clause learning, which prevents revisiting infeasible states, and non-chronological backjumping, which skips unnecessary exploration. Advanced heuristics, such as VSIDS, further improved decision-making, making CDCL the foundation of modern SAT solvers and enabling efficient handling of practical instances.

In this context, the objective of this report is to document the process of implementing a SAT solver based on the CDCL algorithm. The implementation includes parsing DIMACS CNF instances, unit propagation using the two-watched literals technique, and employing the VSIDS heuristic for selecting unassigned variables during the DECIDE process. The remainder of the report is organized as follows: section 2 discusses the implementation details of the solver, section 3 presents the experimental evaluation, and section 4 concludes the report.

2 Implementation

The implementation was done in the **Python** programming language and was inspired by well-known implementations[1] and educational content available on

[1] Python reference implementation and **MiniSat** source code.

the Internet[2], as well as materials used in the SAT solving lessons of the course. The following subsections discuss in more detail each specific aspect of the implementation.

2.1 Classes

To represent the key entities throughout the resolution of a SAT problem, four main classes were used: LITERAL, CLAUSE, FORMULA, and ASSIGNMENT.

Literal. A data class that stores an integer for the literal and a boolean indicating whether it is negated or not. Additionally, this class implements the method NEGATION, which returns a new LITERAL instance with the negation of the variable's negation status.

Clause. A class that stores a clause through a vector of instances of the LITERAL class. It also implements a method called CONTAINS, which checks whether a literal is present in the clause, either as its negation or not.

Formula. A class that stores a formula using a vector of CLAUSE instances.

Assignment. A class that stores information related to the assignments used within the solver. The idea is to have in the solver a set of assignments, $\mathcal{M}$, represented by a vector of ASSIGNMENT instances. Each ASSIGNMENT instance holds the decision level at which it was assigned, the assigned literal, the boolean value assigned to it, and the antecedent clause (if the literal was assigned through unit propagation; if not, this attribute is set to **None**).

2.2 Parser

The implementation of the parser consists of a single function that takes the path to a DIMACS CNF file as a parameter and returns a tuple containing the number of variables in the problem, the number of clauses in the problem, and an instance of the FORMULA type containing all the clauses read from the input. The algorithm 1 presents a pseudocode for the implemented solution.

In general, the parser implementation simply iterates through the lines and processes them based on the prefix present in each line (such as `"c"` or `"p"`) or the absence of a prefix, which indicates a clause line. The only particularity of this function is the vector CURR, which is used to accumulate literals from clauses that span multiple lines in the file. When a line ending with `"0"` is encountered, its literals are appended to CURR, the clause is inserted into the formula, and the vector is reset to empty. When a line ending with a literal is encountered, the vector simply appends the literals from that line and proceeds.

[2] Lessons about SAT solving in the YouTube.

Algorithm 1 Parser for the DIMACS CNF format

```
1: procedure ParseDimacsInstance(I)
2:     Curr ← ∅
3:     N ← 0
4:     M ← 0
5:     φ ← ∅
6:     File ← Open(I, "r")
7:     if not File then
8:         return None
9:     end if
10:    for Line in File do
11:        if not Line then
12:            break
13:        end if
14:        if First(Line) = "c" then
15:            continue
16:        else if First(Line) = "p" then
17:            N, M ← Process(Line)
18:        else
19:            if First(Line) = "0" then
20:                break
21:            end if
22:            LitSet ← ExtractLiterals(Line)
23:            Append(Curr, LitSet)
24:            if EndsWith(LitSet) = "0" then
25:                Append(φ, Curr)
26:                Curr ← ∅
27:            end if
28:        end if
29:    end for
30:    return N, M, φ
31: end procedure
```

2.3 Solver

For the implementation of the solver, three main variables were used:

- $\mathcal{M}$: a list of type Assignment that stores the assignments made during the solving process.
- Assignment: a bit-vector that marks the literals that have been assigned and the truth value associated with them.
- DecisionLevel: a variable that tracks the current decision level during the resolution of a problem.

Two variables were used to manage assignments because each is suited for different operations. The list $\mathcal{M}$ is convenient for storing detailed information about each assignment and benefits from Python's efficient Append and Pop operations. On the other hand, the bit-vector Assignment is useful for quickly

checking whether a literal has been assigned and, if so, determining its truth value in $\mathcal{O}(1)$ time.

In addition, several auxiliary functions were defined to operate on the established structures and solve the problems. While not all implemented functions will be discussed, the most important ones are:

Assign and Unassign. Two functions that were implemented to manage the assignment variables, either performing new assignments or removing existing ones.

SolveOnlyNegativeOrPositiveLiterals. A function was implemented to handle the Pure rule in the CDCL proof system, assigning literals that appear exclusively as positive or negative in the formula's clauses. Once assigned, these literals are added to the propagation queue to be propagated later, updating the watched literals in the affected clauses.

SolveUnitClauses. A function was implemented to traverse the formula, searching for clauses containing only one literal. It assigns the appropriate truth value to the literal to satisfy the clause. If a conflict is detected (an attempt to assign a truth value contrary to the current assignment of a literal), the function returns a conflict status, and the solver concludes the problem as UNSAT.

AllClausesAreSatisfied. A function that was implemented to run in the main loop of the algorithm. It checks whether the current assignment satisfies all clauses by traversing the formula and verifying if, for each clause, there is at least one assigned literal that evaluates to $\top$. If this condition is met, the function returns **True**, and the algorithm terminates, as the problem is determined to be SAT, and a model has been found. Otherwise, the algorithm continues executing by deciding new literals and performing propagations.

ConflictAnalysis. A function that takes the conflicting clause as input, identifies the first Unique Implication Point (UIP) – the earliest decision level literal on the implication graph that causes the conflict – and returns the new clause to be learned along with the new decision level for non-chronological backjumping.

Finding the first UIP is achieved through consecutive iterations where the most recently propagated literal at the current decision level is selected. A resolution step is then applied between the conflicting clause and the antecedent clause, which is the clause that triggered the propagation of the selected literal. This process continues until no literals propagated at the current decision level remain in the clause.

Once the first UIP is found, the algorithm determines the new decision level by selecting the highest decision level among the literals in the learned clause

that is different from the current decision level. This ensures the solver backjumps directly to the most relevant decision level, avoiding unnecessary revisits to irrelevant parts of the search space.

As will be further discussed in subsection 2.4, after determining the learned clause, the activity dictionary used by the VSIDS heuristic is updated, and decay is applied to the literals. Additionally, a specific aspect of this implementation is the use of a SEEN list, which stores the literals that have already been processed. This ensures that repeated literals are not revisited, preventing potential loops and improving the efficiency of the clause learning process. The algorithm 2 presents a pseudocode for the implemented conflict analysis.

Algorithm 2 Conflict analysis procedure to find de first UIP

```
1: procedure ConflictAnalysis(c)
2:     LearnedClause ← GetClause(φ, c)
3:     Seen ← ∅
4:     Literals ← literals in LearnedClause that where propagated in the current
   decision level
5:     while Len(Literals) ≠ 0 do
6:         Literal = Pop(Literals)
7:         if Literal in Seen then
8:             continue
9:         end if
10:        Append(Seen, Literal)
11:        LearnedClause ← Resolution(LearnedClause, Literal)
12:        Literals ← literals in LearnedClause that where propagated in the cur-
   rent decision level
13:    end while
14:    UpdateActivity(LearnedClause)
15:    ApplyDecay()
16:    DecisionLevels ← decision level of the literals in LearnedClause
17:    if Len(DecisionLevels) < 2 then
18:        return 0, LearnedClause
19:    else
20:        return Max(DecisionLevels), LearnedClause
21:    end if
22: end procedure
```

Learn and Backjump.

- LEARN: It takes the conflict clause as a parameter, appends it to the end of the formula, and updates the watched literals of this clause to the most recently assigned literals in the solution (the literals of the clause with the highest decision levels in the assignment).
- BACKJUMP: It takes the new decision level as a parameter and removes from the assignment all literals assigned at decision levels higher than the new

one. Since there are two assignment structures, both are updated: entries in $\mathcal{M}$ are removed, and the unassigned literals in the bit-vector Assignment are set to **None**.

Solve. The main method called by the Main function to solve a SAT problem. It begins with a purification step. During this step, literals from unit clauses or literals that appear only positively or negatively in the formula are propagated.

Then, while not all clauses are satisfied, the algorithm decides a literal and propagates as long as there are literals to propagate. If a conflict is found, it is resolved through conflict analysis. If a conflict is encountered at decision level 0, or during the literal propagation step in the purification phase, the algorithm returns UNSAT.

Eventually, when an assignment that satisfies all the formula's clauses is found, the main loop terminates, and this model is returned, confirming that the formula is SAT. The algorithm 3 presents a pseudocode for the implementation.

Algorithm 3 Main function to solve a SAT problem

```
1: procedure Solve(φ)
2:     ToPropagate ← ∅
3:     ToPropagate ← SolveOnlyNegativeOrPositiveLiterals()
4:     ToPropagate ← SolveUnitClauses()
5:     if UnitPropagation(ToPropagate) leads to a conflict then
6:         return UNSAT
7:     end if
8:     while not AllClausesAreSatisfied() do
9:         Decide and assign a Literal
10:        Append(ToPropagate, Literal)
11:        while True do
12:            if UnitPropagation(ToPropagate) leads to a conflict then
13:                c ← GetConflictClause()
14:                if DecisionLevel = 0 then
15:                    return UNSAT
16:                end if
17:                NewDecisionLevel, LearnedClause ← ConflictAnalysis(c)
18:                Learn(LearnedClause)
19:                Backjump(NewDecisionLevel)
20:                Assign and propagate the unassigned literal in the LearnedClause
21:            end if
22:        end while
23:    end while
24:    return M
25: end procedure
```

2.4 Optimizations and heuristics

This section provides a more detailed discussion of the optimizations implemented in the code, focusing particularly on the implementation of two-watched literals and the VSIDS heuristic.

Two-watched literals. The implementation of the two-watched literals occurs in two stages. The first stage involves defining the structure, which, in this case, is a dictionary WatchedLiterals indexed by clauses. This dictionary stores the literals currently being watched by each clause.

The second stage takes place in the unit propagation function. In it, while there are literals to propagate, the following steps are performed: the algorithm retrieves the clauses watching the negation of the literal being propagated. These clauses are potentially falsified due to the current assignment. For each such clause, the algorithm iterates through its literals, searching for one that is either unassigned or evaluates to $\top$, in order to update the watched literal.

If it is not possible to update the watched literal (i.e., all non-watched literals evaluate to $\bot$), the algorithm examines the second watched literal of the clause. If no second watched literal exists, this indicates a unit clause and a conflict is reported. If the second watched literal exists and is unassigned, it is assigned a truth value, and the literal is added to the propagation queue. If the second watched literal is already assigned, its truth value is checked: if $\top$, no action is taken; if $\bot$, a conflict is reported.

In both cases where a conflict occurs, the clause in which the conflict was detected is returned. The algorithm 4 presents a pseudocode for the implementation.

VSIDS. To implement the Variable State Independent Decaying Sum (VSIDS) heuristic, three main variables were used:

- Activity: a dictionary mapping each Literal to a float, representing its activity level.
- DecayFactor: a constant representing the decay factor.
- Increment: a value indicating how much the activity of each literal appearing in the conflict clause should increase.

Initially, the Activity dictionary is populated by iterating through all the clauses in the formula and increasing the activity value for each literal that appears in the formula. In addition to that, the activity of a literal is increased whenever it appears in the learned clause during conflict analysis. After the first UIP (Unique Implication Point) is found and the learned clause is defined, the literals in that clause are iterated over, and their activity values in the Activity dictionary are increased accordingly.

The ApplyDecay function was implemented and is called after increasing the activity of literals in the learned clause. This function essentially iterates through the activity dictionary and multiplies each literal's activity value by the

Algorithm 4 Unit propagation procedure

```
1: procedure UnitPropagation(ToPropagate)
2:     while Len(ToPropagate) ≠ 0 do
3:         NegatedLit ← Pop(ToPropagate).Negation()
4:         Clauses ← WatchedLiterals[NegatedLit]
5:         for c in Clauses do
6:             Literals ← GetLiterals(c)
7:             for lit in Literals do
8:                 Literals ← GetLiterals(c)
9:                 if c in WatchedLiterals[NegatedLit] then
10:                    continue
11:                end if
12:                if lit is not assigned or evaluates to ⊤ then
13:                    Remove(WatchedLiterals[NegatedLit], c)
14:                    Append(WatchedLiterals[lit], c)
15:                    break
16:                end if
17:            end for
18:            if no pointer was updated then
19:                if c has only one watched literal then          ▷ c is an unit clause
20:                    return 1, c
21:                end if
22:                OtherWatchedLiteral ← the other watched literal of c
23:                if OtherWatchedLiteral not in M then
24:                    Assign(OtherWatchedLiteral)
25:                    Append(ToPropagate, OtherWatchedLiteral)
26:                else
27:                    if OtherWatchedLiteral evaluates to ⊤ then
28:                        continue
29:                    else
30:                        return 1, c
31:                    end if
32:                end if
33:            end if
34:        end for
35:    end while
36:    return 0, None
37: end procedure
```

DecayFactor, ensuring that literals frequently involved in conflicts retain higher activity values compared to those that are no longer as relevant.

Finally, the Activity dictionary is used during the Decide process, where unassigned literals are retrieved and sorted by their activity levels. The literal with the highest activity value is then selected for assignment. Since the dictionary differentiates between a literal and its negation, selecting a literal also determines the truth value that will be assigned to it.

3 Evaluation

The implemented solver will be compared with the well-established SAT solver **MiniSat**. Two main aspects will be considered: the *correctness* of the implementation, ensuring it returns SAT for satisfiable instances and UNSAT for unsatisfiable ones, and the *performance*, where the primary metric will be the time taken to solve the problem.

The benchmark set used for evaluation was provided by the professor on the course's website. It consists of 24 SAT instances and 25 UNSAT instances, varying in the number of variables and clauses, as well as whether they represent the encoding of another problem in propositional logic or are artificially generated.

The experiments were run on an Intel i5-10210U (8) @ 4.200GHz with 20 GB of DDR4 RAM running Ubuntu 24.04 LTS, with a timeout limit of 20 minutes for each problem.

3.1 Correctness

After running the benchmarks, the implemented solver timed out on two instances: `prime1849.cnf`, among the SAT instances, and `cnfgen-ram-4-3-10.cnf`, among the UNSAT instances. For the remaining instances, the results obtained by the implementation aligned with the satisfiability of each instance. In this sense, the implemented solver demonstrated correctness with respect to the expected outcomes.

Although additional individual tests were conducted and supported this conclusion, an interesting next step would be to verify the soundness and completeness properties of the solver using a larger and more robust set of instances. This would provide greater confidence in the implementation.

3.2 Performance

Regarding performance, the cactus plots in figure 1 summarize the results obtained from running both solvers: the implementation developed in this project and **MiniSat**. As observed, the implemented solver initially performs relatively well and competitively compared to **MiniSat**. However, when faced with more complex instances or those with a larger number of clauses, **MiniSat** significantly outperforms it.

This discrepancy can be attributed to several factors, such as the implementation language (**MiniSat** is written in C, which is well-known for its superior performance compared to Python), the use of additional heuristics, like restart techniques, which were not implemented in this project's solution, and other optimizations in the solver's internal structures. These optimizations could also be applied to the project's implementation but were not included due to a lack of familiarity with alternative approaches to accomplish the same tasks.

Despite this, the results are promising, and the ideas for potential improvements to the solver provide a clear direction for changes needed to make it more competitive with well-established alternatives such as MiniSat, Glucose, CaDiCaL, and many others.

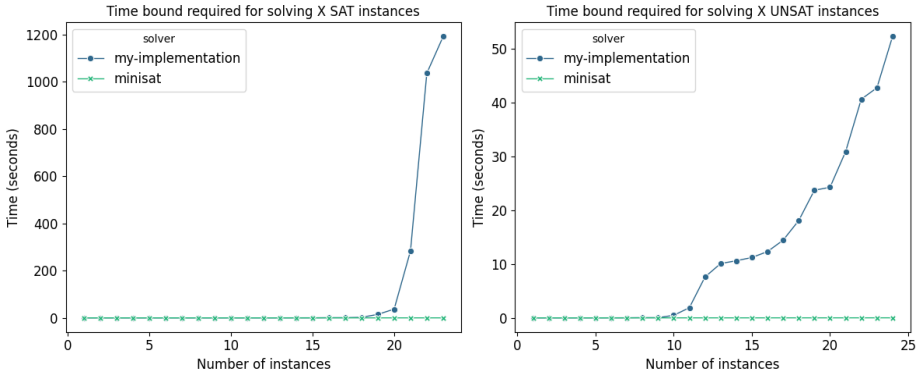

Fig. 1. Cactus plot for solving the SAT problems in the benchmark. Each point represents the runtime for one instance, with the x-axis showing the number of instances and the y-axis indicating the maximum runtime within a given time bound.

4 Conclusion

The implementation of the CDCL-based SAT solver provided a solid understanding of its architecture and effectiveness in solving SAT problems. Breaking the solver into components like clause learning, non-chronological backjumping, unit propagation with two-watched literals, and heuristics such as VSIDS highlighted the importance of each technique and how they work together to improve efficiency.

Despite its benefits, the implementation posed challenges, including managing complex data structures like the watched literals dictionary and assignment variables, debugging conflicts, and integrating learned clauses. Addressing these difficulties was key to grasping the complexity of modern SAT solvers and the sophistication of their algorithmic strategies.

References

1. Cook, S.A.: The complexity of theorem proving procedures. In: Proceedings of the Third Annual ACM Symposium. pp. 151–158. ACM, New York (1971)
2. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM **5**(7), 394–397 (1962)
3. Marques-Silva, J., Sakallah, K.: GRASP – a new search algorithm for satisfiability. In: Int'l Conf. on CAD. pp. 220–227 (1996)