# ECE 51216: Implementation of SAT Solver

Vishal Kevat, Adrija Chakraborty
Group 5

May 9, 2025

**Motivation**

Boolean Satisfiability (SAT) is the problem of determining whether there exists an assignment of truth values to variables that makes a given Boolean formula evaluate to true. As the first problem proven to be NP-complete, SAT is a cornerstone of theoretical computer science and has broad practical relevance across various domains, including Electronic Design Automation (EDA), software verification, and artificial intelligence. In the context of EDA, SAT solvers are indispensable tools used in a range of applications such as formal verification, equivalence checking, timing and power analysis, automatic test pattern generation, and design debugging.

The core of many industrial-grade SAT solvers is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, a backtracking-based search method for deciding satisfiability. Over the years, the DPLL framework has been enhanced with sophisticated heuristics such as Conflict-Driven Clause Learning (CDCL), which allows the solver to learn from conflicts and Variable State Independent Decaying Sum (VSIDS), a dynamic variable selection heuristic that improves search efficiency by prioritizing the most active variables.

The objective of this project is to design and implement a SAT solver in C++, aiming for high performance and low memory usage. Our solver is based on the DPLL algorithm and incorporates both CDCL and VSIDS to accelerate the solving process. Beyond implementation, we have conducted a detailed analysis of the solver's functionality, evaluated its runtime performance across varying problem sizes and studied the effects of enabling or disabling individual heuristics.

# 1 Introduction

## 1.1 Satisfiability in Boolean Algebra

A Boolean formula $\psi$ is said to be in *conjunctive normal form* (CNF) if it is expressed as a conjunction of $k$ clauses $C_1, C_2, \ldots, C_k$, where each clause is a disjunction of one or more *literals*. A literal is defined as either a Boolean variable $x_i$ or its negation $\neg x_i$, for $i = 1, 2, \ldots, n$. Each CNF formula corresponds to a unique Boolean function $f(x_1, x_2, \ldots, x_n)$, and each clause in the formula can be viewed as a logical constraint that restricts the valid assignments to the variables.

Multiple CNF formulas can represent the same Boolean function, as the transformation to CNF is not unique. The *Boolean Satisfiability problem* (SAT) asks whether there exists a truth assignment to the variables $x_1, x_2, \ldots, x_n$ such that the entire CNF formula $\psi$ evaluates to true, i.e., whether $f(x_1, x_2, \ldots, x_n) = 1$. If no such assignment exists, the formula is said to be unsatisfiable (UNSAT), meaning the function evaluates to false for all inputs.

In the context of search algorithms for SAT, variables can be assigned a logic value, either 0 or 1. Alternatively, variables may also be unassigned. Given an assignment $m$, if all variables are assigned a value in $\{0, 1\}$, then $m$ is referred to as a complete assignment. Otherwise it is a partial assignment [1].

The SAT (Satisfiability) problem involves finding such an assignment that satisfies a given CNF formula. The problem is classified as SAT if a satisfying assignment exists, and UNSAT if no such assignment exists. Figure 1 represents a CNF which has positive literal, negative literal and a clause.

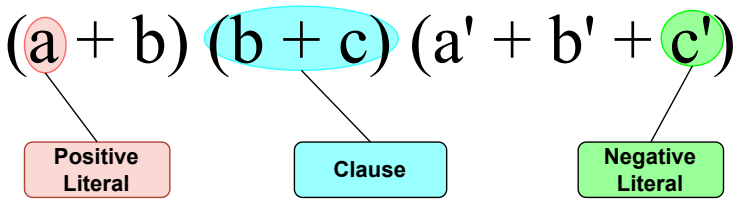


Figure 1: CNF with Clause, Positive and Negative literal

The SAT problem plays a pivotal role in both theoretical computer science and practical technology domains, primarily due to its effectiveness in ensuring the correctness, reliability, and efficiency of complex systems. Its widespread application in verifying the integrity of both software and hardware systems highlights its foundational importance in advanced computational tasks, formal methods, and research in automated reasoning.

## 1.2 DPLL

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm is a fundamental procedure for solving the Boolean Satisfiability (SAT) problem. It is a complete, backtracking-based search algorithm used to determine the satisfiability of propositional logic formulas expressed in Conjunctive Normal Form (CNF).

The algorithm works by recursively selecting a variable, assigning it a truth value, and simplifying the formula accordingly. If a conflict is detected—i.e., a clause becomes unsatisfiable—the algorithm backtracks and tries alternative assignments. DPLL incorporates two key techniques: *unit propagation*, where unit clauses enforce necessary assignments, and *pure literal elimination*, which allows the assignment of literals that appear with only one polarity.

DPLL serves as the foundation for modern SAT solvers and is typically extended with advanced features such as *conflict-driven clause learning* (CDCL) and *variable state independent decaying sum* (VSIDS) to significantly improve performance on large and complex SAT instances. Figure 2 illustrates the working of DPLL algorithm.

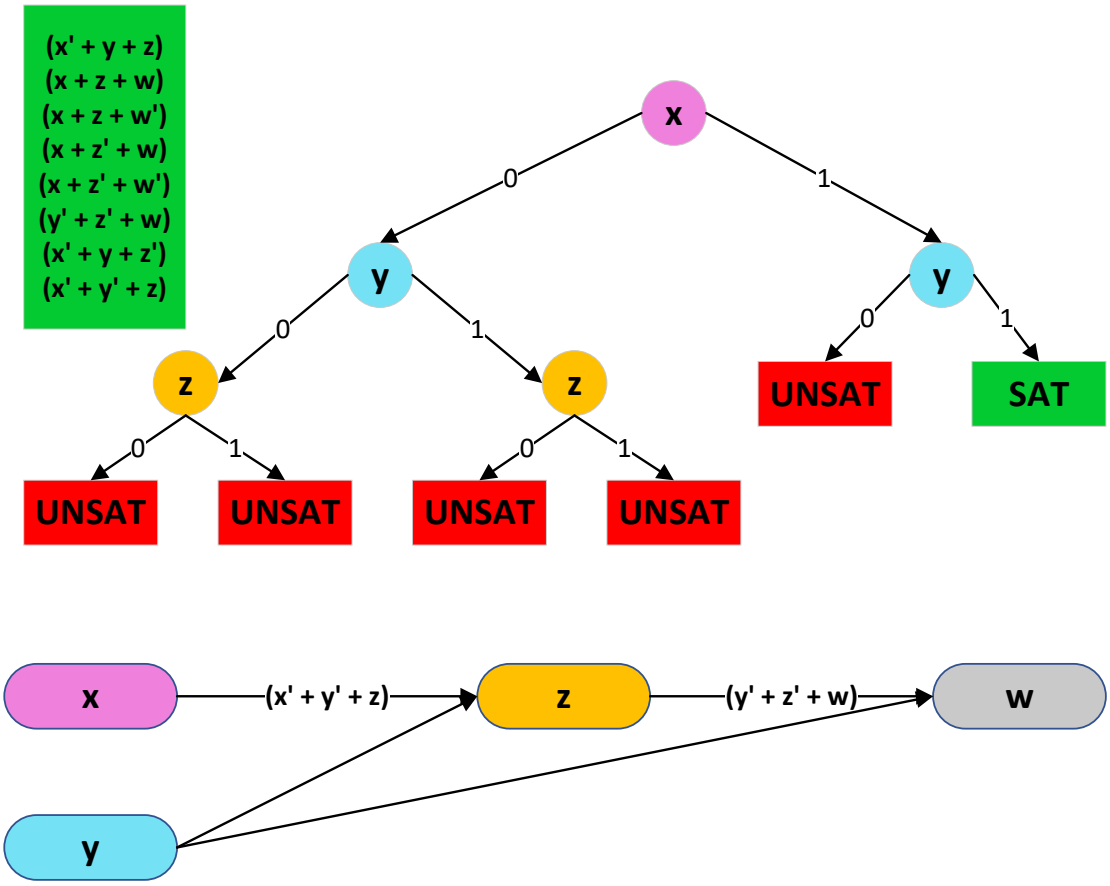


Figure 2: Solving CNF formula using basic DPLL

## 1.3 Conflict-Driven Clause Learning (CDCL)

Given a CNF formula $\psi$, a CDCL-based SAT solver attempts to find a satisfying assignment for $\psi$. The search proceeds in three main steps: *decision making* or *branching*, where the solver heuristically chooses a variable and assigns it a truth value; *propagation*, where the consequences of the assignment are deduced (typically using Boolean Constraint Propagation, BCP, which repeatedly applies unit clause rules to deduce implied assignments); and *conflict analysis*, which is triggered when a conflict (i.e., a contradiction) is detected [2].

Conflict-Driven Clause Learning (CDCL) is an advanced technique used to enhance the efficiency of SAT solvers. When a conflict is encountered during the search, the solver analyzes

the conflict to determine its root cause. Based on this analysis, it learns a new clause—called a *conflict clause*—that prevents the same conflict from occurring again in the future. This learned clause is then added to the formula, effectively pruning the search space. Additionally, the solver performs a non-chronological backtrack, jumping back to an earlier decision level that is relevant to the conflict, rather than simply undoing the most recent decision. This combination of learning and intelligent backtracking makes CDCL solvers significantly more powerful than normal DPLL-based methods.

### 1.4 Variable State Independent Decaying Sum (VSIDS)

Variable State Independent Decaying Sum (VSIDS) refers to a family of branching heuristics widely used in modern CDCL SAT solvers. These heuristics dynamically rank all variables of a Boolean formula during the run of the solver. The key idea is to collect statistics from learnt clauses to guide the direction of the search, with a preference for variables that appear frequently in recent conflicts.

A central feature of VSIDS is its scoring mechanism: each variable is assigned a numerical activity score, which is increased (*additively bumped*) whenever the variable appears in a learnt clause. Over time, all activity scores are periodically scaled down (*multiplicatively decayed*), which gives more weight to variables involved in recent conflicts and gradually forgets older activity. This prioritizes variables that are currently more "relevant" to the unsatisfiability reasoning, allowing the solver to adapt its branching decisions to the evolving structure of the search space.

This combination of additive bumping and multiplicative decay leads to a form of temporal locality, effectively capturing which parts of the formula are currently active or critical for finding a satisfying assignment or proving unsatisfiability. Despite its simplicity, VSIDS has been remarkably successful and is a key component in the efficiency of state-of-the-art SAT solvers [3].

## 2 Implementation

### 2.1 Input Parsing

In our project, Boolean expressions are represented in Conjunctive Normal Form (CNF), where each formula is a conjunction of clauses, and each clause is a disjunction of literals. The CNF file follows the DIMACS standard format. Literals are encoded as integers: a positive integer $n$ represents the Boolean variable $x_n$, while a negative integer $-n$ represents its negation $\neg x_n$. Each clause is written on a separate line, with literals separated by spaces and terminated by a `0`. Lines beginning with the character `c` are treated as comments, and lines beginning with `p` provide problem metadata (such as the number of variables and clauses). Additional non-informative lines starting with `%` or `0` are also ignored by the parser.

The CNF formula is parsed using a helper function that reads the file line by line. For

each relevant line (i.e., a clause line), the function splits the line into a list of integers. The trailing `0` is discarded, and the remaining integers are inserted into a set, representing a single clause. Each clause is thus stored as an `unordered_set<int>` to ensure that duplicate literals within a clause are ignored. The collection of all clauses is stored as a `vector` of these sets, corresponding to the full CNF formula. This structured representation is then used as input to the DPLL-based SAT solver to determine the satisfiability of the given formula.

## 2.2 Pseudo Code

**Main Function:** The `main` function serves as the entry point for executing the SAT solver. It expects exactly one command-line argument: the path to a CNF-formatted input file. Upon receiving a valid file path, it invokes the `load_cnf` function to parse the file into a list of clauses, each represented as a set of integers. These clauses are passed to an instance of `DPLLSolver_DS`, which encapsulates the SAT solving process using the DPLL algorithm enhanced with CDCL and the VSIDS branching heuristic. After solving, it outputs whether the formula is satisfiable (`SAT`) or not (`UNSAT`), and if satisfiable, it prints the variable assignments found. Overall, the function orchestrates file parsing, SAT solving, and performance reporting in a clean and efficient manner.

**Algorithm 2.1:** Main Function: SAT Solver Entry Point

```
1  FUNCTION main(argc, argv):
2  → Check for a valid cnf file
3  // Load and parse CNF clauses
4  → Call the input parser to obtain the list of clauses
5  // Initialize solver
6  → Create instance of DPLLSolver_DS with parsed clauses
7  // Solve the SAT problem
8  → Call solver.solve() and store the result
9  // Record and report results
10 → Print SAT/UNSAT result
11 if result is SAT then
12     Print variable assignments
13 end if
```

**Class Initialization:** The `DPLLSolver_DS` class implements a SAT solver based on the DPLL algorithm, incorporating CDCL and VSIDS heuristics. The constructor initializes the solver with a given CNF formula by copying the clauses into both a mutable working set (`cnf`) and an immutable reference copy (`original_cnf`), while initially marking all clauses as unsatisfied. It also determines the highest variable index to properly size internal data structures, and maps each variable to the clauses in which it appears, distinguishing between positive and negative occurrences.

Some of the important parameters include `learned_clause_limit_percentage`, which controls the maximum number of learned clauses allowed as a percentage of the original formula

size, and `max_learned_clause_len`, which limits the size of learned clauses to prevent overly complex reasoning. Additionally, the constructor initializes VSIDS-related counters such as `conflict_weight`, `decay_factor`, and `decay_interval`, which govern how literal activity scores are updated and influence the branching decisions during the solving process. This initialization equips the solver with a robust setup for efficient decision-making, propagation, conflict analysis, and clause learning.

**Algorithm 2.2:** DPLLSolver_DS - Basic Functionality

**CLASS** `DPLLSolver_DS`(cnf, learned_clause_limit_percentage, max_learned_clause_len):
// Initialization
→ Copy input `cnf` into mutable and original clause sets
→ Setup variable metadata, assignments, and clause tracking structures
→ Configure learning and VSIDS heuristic parameters
// Entry point: begin search with empty assignment
**solve()**:
→ Clear assignments stack
→ Invoke recursive `dpll()` procedure
// The main recursive DPLL algorithm.
**dpll()**:
→ Perform unit propagation and pure literal elimination
→ Choose branching literal using VSIDS
→ Recurse with assignments; backtrack on failure
→ Returns True or False depending on satisfiability of the formula
// Updating literal assignment
**assign(lit)**, **unassign(lit)**:
→ Apply or revert a literal assignment, updating affected clauses
// Add a learned clause derived from a conflict.
**add_learned_clause(clause)**:
→ Add clause from conflict, update clause lists and counters
// print the assignments
→ The **main()** function calls **solve()** and prints the final result

**Solve Function:** This method resets any prior assignments and initiates the DPLL recursive procedure to find a solution.

**Algorithm 2.3:** Solve Function

**Function** `solve`():
// Clear assignments stack and reset solver state
→ Clear assignments_stack
→ Initialize or reset the solver state
// Start the DPLL procedure
→ Start the DPLL procedure

**DPLL Algorithm:** The core of our SAT solver is the DPLL (Davis-Putnam-Logemann-Loveland) algorithm, implemented using a recursive search strategy combined with Boolean Constraint Propagation (BCP). Key components of this approach include unit clause propagation, which simplifies the formula by assigning forced truth values; pure literal elimination, which removes variables that appear with only one polarity; and Conflict-Driven Clause Learning (CDCL), which prevents repeated conflicts. Additionally, we employ the VSIDS heuristic to guide decision-making strategically and enhance the efficiency of the search process.

**Algorithm 2.4:** DPLL Algorithm for SAT

```
Function dpll():
// Update decision counters
→ Increment decision_count; apply decay if needed
// Unit Clause Propagation
while unit clauses exist do
    → Assign unit literal; if conflict, learn clause and return false
    → Push to stack; update unsatisfied_clauses
end while
if all clauses satisfied then return true
end if
// Pure Literal Elimination
while pure literals exist do
    → Assign; if conflict, return false
    → Push to stack
end while
if all clauses satisfied then return true
end if
// Branching
→ Select literal using VSIDS
for val in {lit, −lit} do
    if assign(val) then
        → Push; recurse dpll(); if success, return true
        → Backtrack
    else
        → Learn clause if conflict; return false
    end if
end for
// No solution
→ Return false
```

**Assign Function:** The Assign function is pivotal in our SAT solver. It attempts to assign a value to a variable and updates the formula accordingly. This function plays a crucial role in managing the solver's state by detecting conflicts and modifying the clauses based on the results of the assignment.

**Algorithm 2.5:** Assign Literal Function

**Function** `assign`(*literal*):
→ Assign the corresponding value to the variable represented by *literal*
// Clause Updates After Assignment
→ Remove all clauses satisfied by this assignment from the list of unsatisfied clauses
→ Remove the negated form of the assigned literal from all remaining unsatisfied clauses.
// Conflict Detection and Clause Learning
→ Check for any conflicts resulting from the assignment
→ If a conflict is found, construct a conflict (learned) clause
→ Record the conflict clause and update relevant data structures
→ Return the outcome of the assignment (success or conflict)

**Unassign Function:** The Unassign function plays a crucial role in the SAT solver's backtracking mechanism. It reverses the assignment of a variable, restoring its previous value and reverting any clause modifications resulting from that assignment. This restoration enables the solver to backtrack, re-evaluate previously unsatisfied clauses, and explore alternative branches in the search space for a satisfying solution.

**Algorithm 2.6:** Unassign Literal Function

**Function** `unassign`(*literal*):
→ Reset the variable corresponding to *literal* to an unassigned state
// Restore clause information affected by this unassignment
→ Reintroduce the negated literal into clauses from which it was previously removed
→ Reconsider any clauses previously marked as satisfied due to this literal
→ Update the set of unsatisfied clauses accordingly

**Add Learned Conflict Clause:** This function enhances the SAT solver's efficiency by integrating conflict-driven clause learning. By incorporating clauses derived from previous conflicts, the solver can proactively avoid repeating the same mistakes, thereby pruning the search space and improving convergence. Additionally, the weight of literals appearing in learned conflict clauses is increased, guiding future decisions toward literals that are more influential in resolving conflicts.

**Algorithm 2.7:** Add Learned Clause Function

**Function** `add_learned_clause`(*clause*):
→ Insert the learned clause into both `cnf` and `original_cnf`, if applicable
// Update internal data structures based on the new conflict clause
→ Associate the clause with relevant variables and update clause tracking structures
→ Mark the clause as unsatisfied (if not already satisfied)
→ Increment the total count of learned clauses
→ Invoke `boost_conflict_literals()` to increase weights of literals in the learned clause

**Decay Counters:** The `decay_counters` function reduces (or 'decays') the influence of previous conflict activity for each variable by multiplying their positive and negative conflict counts by a decay factor (a number less than 1). This gradually reduces the weight of older conflicts, allowing more recent conflict activity to have a stronger influence on which variables are chosen next, helping the solver adapt to current conditions in the search space.

**Algorithm 2.8:** Decay Counters Function

1 **Function** `decay_counters`():
2 → For each variable, multiply its VSIDS counters by the decay factor
3 → Apply decay to both `pos_count` and `neg_count` to gradually reduce the influence of older conflicts

**Boost Conflict Literals:** This function increases the activity scores (heuristic counters) of literals that appear in conflict clauses. By boosting their values, the solver prioritizes these literals in future decisions, improving the likelihood of quickly resolving similar conflicts.

**Algorithm 2.9:** Boost Conflict Literals Function

1 **Function** `boost_conflict_literals`(*clause*):
2 → For each literal in the learned conflict clause, increment its VSIDS activity counter by a predefined conflict weight
3 → Apply decay to all counters immediately afterward to maintain relative prioritization and prevent overflow

# 3 Data Structures Used in DPLL Solver

## 3.1 Dynamic and Original CNF Representations

**Structure:** `vector<unordered_set<int>> cnf`,
`vector<unordered_set<int>> original_cnf`

**Purpose:** These two data structures store the clauses of the Boolean formula in Conjunctive Normal Form (CNF). `original_cnf` retains the initial clauses along with any learned clauses during conflict analysis, while `cnf` is dynamically updated throughout the solving process (e.g., literals are removed upon assignment). This separation allows restoration and accurate tracking of learned clauses and supports efficient backtracking.

## 3.2 Variables Dictionary

**Structure:** `vector<VariableInfo> variables`

**Purpose:** Each element in this vector corresponds to a Boolean variable and holds its assignment status, clause membership lists (positive and negative appearances), and VSIDS

activity counters. This structure enables quick access to the state of variables, facilitating efficient propagation and heuristic decision-making throughout the execution of the DPLL algorithm.

## 3.3 Assignments Stack

**Structure:** `vector<int> assignments_stack`

**Purpose:** This stack tracks the sequence of literal assignments made during the solving process. It enables efficient backtracking by reversing recent assignments and plays a critical role in maintaining the current partial model.

## 3.4 Unsatisfied Clause Tracker

**Structure:** `unordered_set<int> unsatisfied_clauses`

**Purpose:** Maintains indices of CNF clauses that are not yet satisfied under the current variable assignments. This helps quickly identify unit clauses, perform conflict detection, and minimize clause scanning during decision and propagation phases.

## 3.5 Conflict Analysis Status

**Structure:** `AssignmentStatus last_assignment_status`

**Purpose:** This structure flags conflicts and stores the reason (a set of literals) leading to a conflict. It supports clause learning by supplying the relevant literals used to construct a new clause upon encountering a conflict.

## 3.6 VSIDS Heuristic Counters

**Structure:** `pos_count`, `neg_count` inside `VariableInfo`

**Purpose:** These floating-point counters record the historical frequency of each variable appearing in conflict clauses, separated by polarity. They support the VSIDS heuristic to prioritize variables for branching based on their activity.

## 3.7 Learned Clause Limit Parameters

**Structure:** `max_learned_clause_len`, `max_learned_clauses`, `learned_clauses_count`

**Purpose:** These integer values control the size and number of learned clauses stored during the solving process to prevent memory blow-up and maintain solver efficiency.

## 3.8 DPLL Solver Parameters

**Structure:** `decay_factor`, `conflict_weight`, `decay_interval`, `decision_count`

**Purpose:** The `decay_factor` controls the rate at which conflict counters decay, gradually reducing the influence of older conflicts to allow more recent ones to dominate. The `conflict_weight` determines the strength of the impact when a literal is involved in a conflict clause, influencing future literal selection during branching. The `decay_interval` defines how often the decay of conflict counters is triggered based on the number of decisions made, ensuring that decay occurs at appropriate intervals. Finally, the `decision_count` keeps track of the total number of decisions, helping to trigger decay and monitor the solver's progress.

A summary of the data structures and data members is given in Figure 3

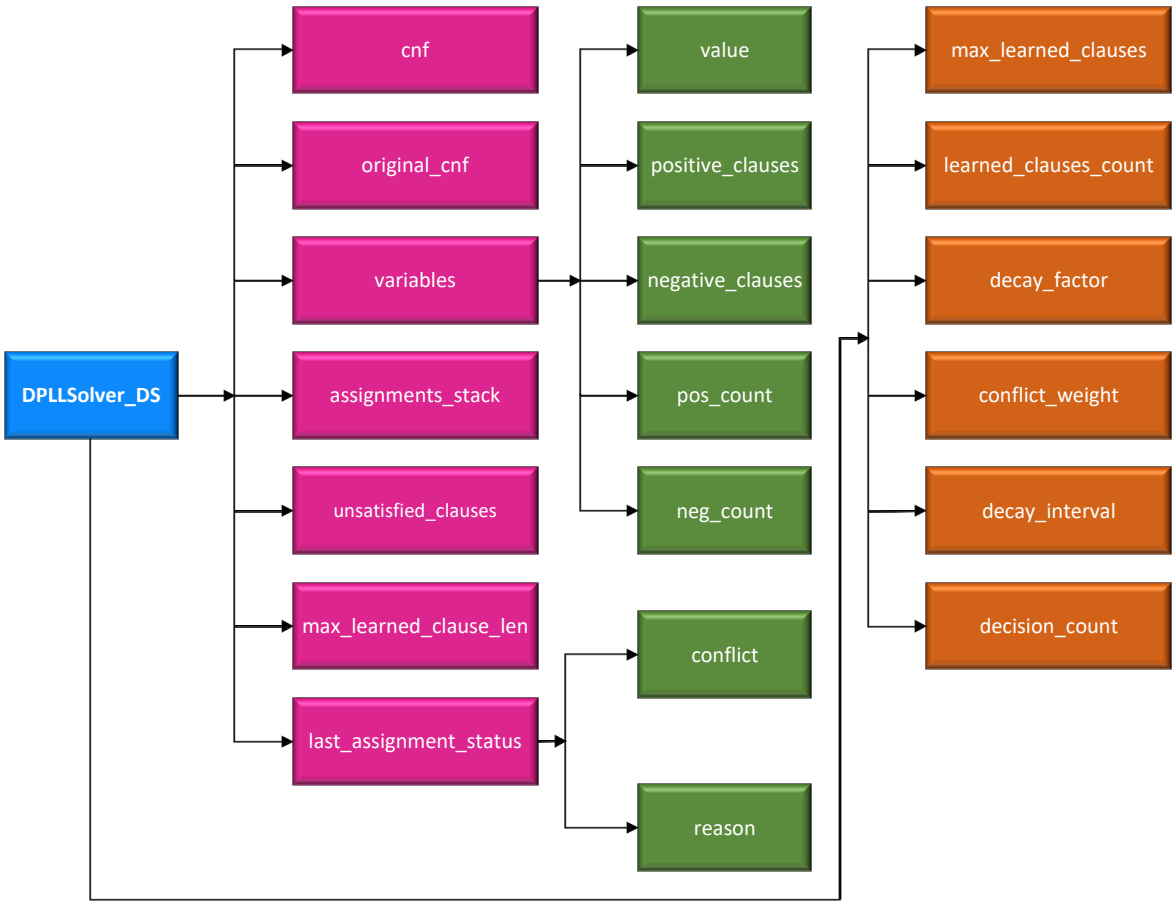


Figure 3: Data Structures and Data Members

# 4 Validation

## 4.1 Experimental Methodology

The goal of our experimental evaluation was to assess the performance of the DPLL solver augmented with CDCL and the VSIDS heuristic. We tested the solver's ability to handle SAT instances of varying complexities, clause counts, and variable sizes.

We first implemented and evaluated a baseline DPLL solver, testing it rigorously on diverse benchmark cases, including edge cases, to measure its time and memory usage. This established a reference point for performance.

Subsequently, CDCL was integrated, and the solver was re-evaluated on the same benchmarks to observe improvements in efficiency. Finally, we incorporated the VSIDS heuristic into the DPLL + CDCL framework and repeated the tests. The results were analyzed to quantify the impact of each enhancement.

## 4.2 Benchmarks & Results

Table 1: Time Taken and Memory Usage for Various CNF

| Benchmark | Variables | Clauses | Result | Time Taken (seconds) | | | Memory Usage (KB) | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | DPLL | CDCL | VSIDS | DPLL | CDCL | VSIDS |
| uf20-0114.cnf | 20 | 91 | SAT | 0.000389 | 0.000107 | 0.000045 | 3760 | 1948 | 1936 |
| uf20-0159.cnf | 20 | 91 | SAT | 0.000781 | 0.000167 | 0.000083 | 3660 | 1932 | 1908 |
| uf20-0518.cnf | 20 | 91 | SAT | 0.000169 | 0.000104 | 0.000036 | 3708 | 1912 | 1908 |
| uf20-0681.cnf | 20 | 91 | SAT | 0.000095 | 0.000117 | 0.000037 | 1944 | 1956 | 1892 |
| uf20-0771.cnf | 20 | 91 | SAT | 0.000176 | 0.000188 | 0.000105 | 3708 | 1988 | 1956 |
| uf20-0941.cnf | 20 | 91 | SAT | 0.000383 | 0.000119 | 0.000048 | 3784 | 1984 | 1940 |
| aim-50-2-0-yes1-4.cnf | 50 | 100 | SAT | 0.212331 | 0.000273 | 0.000317 | 4048 | 3800 | 3660 |
| aim-50-3-4-yes1-4.cnf | 50 | 170 | SAT | 0.010284 | 0.000226 | 0.000104 | 3912 | 3612 | 3624 |
| uuf50-0152.cnf | 50 | 218 | UNSAT | 0.07464 | 0.002649 | 0.002566 | 4108 | 3524 | 3604 |
| uuf50-057.cnf | 50 | 218 | UNSAT | 0.007386 | 0.0031 | 0.001606 | 4016 | 3660 | 3696 |
| uuf50-063.cnf | 50 | 218 | UNSAT | 0.023096 | 0.003222 | 0.001395 | 4156 | 3524 | 3724 |
| uf75-01.cnf | 75 | 325 | SAT | 0.003371 | 0.00149 | 0.000363 | 4632 | 3864 | 3652 |
| uf75-011.cnf | 75 | 325 | SAT | 0.203612 | 0.023379 | 0.002541 | 5068 | 3736 | 3928 |
| uf75-012.cnf | 75 | 325 | SAT | 0.272922 | 0.002886 | 0.017739 | 4916 | 3768 | 3572 |
| uuf75-013.cnf | 75 | 325 | UNSAT | 0.512505 | 0.032258 | 0.014496 | 4944 | 3612 | 3652 |
| flat30-100.cnf | 90 | 300 | SAT | 0.00053 | 0.000172 | 0.000416 | 4056 | 3732 | 3500 |
| flat30-31.cnf | 90 | 300 | SAT | 0.000748 | 0.000172 | 0.000331 | 3836 | 3720 | 3620 |
| flat30-45.cnf | 90 | 300 | SAT | 0.000552 | 0.000154 | 0.000296 | 3968 | 3692 | 3536 |
| flat30-83.cnf | 90 | 300 | SAT | 0.000589 | 0.000161 | 0.000555 | 3968 | 3680 | 3904 |
| uf100-01.cnf | 100 | 430 | SAT | 3.888008 | 0.113287 | 0.004224 | 5664 | 3960 | 4056 |
| uf100-0105.cnf | 100 | 430 | SAT | 0.512649 | 0.073722 | 0.000895 | 5588 | 3856 | 3860 |
| uf100-035.cnf | 100 | 430 | SAT | 1.541247 | 0.092594 | 0.001141 | 5888 | 3852 | 3912 |
| uf100-046.cnf | 100 | 430 | SAT | 14.878872 | 0.106065 | 0.003291 | 6112 | 3992 | 3788 |
| uuf125-027.cnf | 125 | 538 | UNSAT | 42.858711 | 0.948045 | 0.795685 | 6784 | 4036 | 3884 |
| uuf125-064.cnf | 125 | 538 | UNSAT | 41.621994 | 0.867062 | 0.146428 | 7020 | 3980 | 4204 |
| uf150-01.cnf | 150 | 645 | SAT | 30.891334 | 0.212576 | 0.03513 | 8276 | 4144 | 4276 |
| uuf150-067.cnf | 150 | 645 | UNSAT | 307.442699 | 4.421082 | 2.438081 | 8048 | 4240 | 4276 |

Our SAT Solver was rigorously evaluated on over 2,000 benchmark instances sourced from standard repositories such as SATLIB and DIMACS. These benchmarks were used to test the basic DPLL algorithm as well as its enhanced versions with advanced heuristics. Table 1 presents a detailed performance analysis across this diverse set of problems.

In the table, "DPLL" represents the baseline solver using the standard DPLL algorithm, "CDCL" refers to the integration of Conflict-Driven Clause Learning with DPLL, and "VSIDS" denotes the inclusion of the VSIDS heuristic on top of DPLL + CDCL. Advanced heuristics were incrementally integrated to evaluate their individual and combined impact on solver performance.

The benchmark data spans a wide range of variable and clause sizes, offering diverse test conditions. To enable fair comparison, the results have been normalized and visualized through graphs in the following section, effectively highlighting the solver's performance and scalability across varying levels of problem complexity.

# 5 Analysis of Results

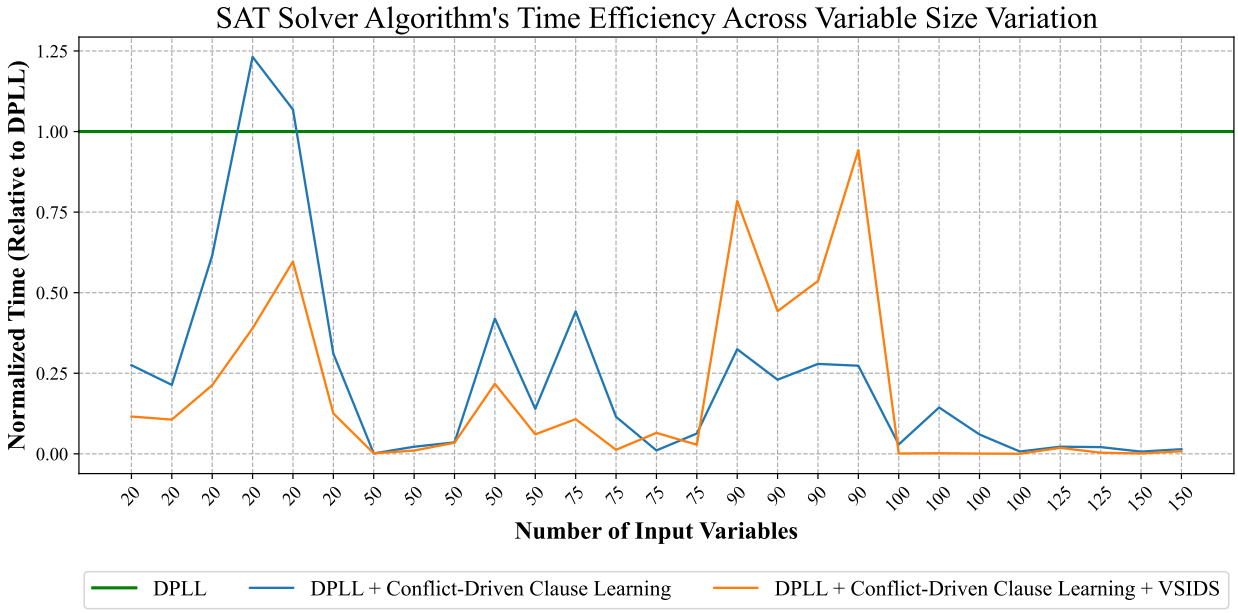


Figure 4: Time Efficiency for Various Heuristics

Figure 4 illustrates the time efficiency of the enhanced SAT solver variants relative to the baseline DPLL algorithm across varying numbers of input variables. The DPLL + Conflict-Driven Clause Learning (blue line) generally outperforms the basic DPLL (green line), especially as problem size increases, showing reduced normalized time in most cases. The addition of VSIDS (shown by orange line) further improves performance in several instances, particularly for medium-to-large input sizes, where it achieves the lowest normalized time. However,

for a few lower variable counts, DPLL + CDCL temporarily shows increased overhead due to the added complexity of conflict analysis. Overall, both enhancements—especially the inclusion of VSIDS—significantly improve solver efficiency and scalability across diverse SAT problem sizes.

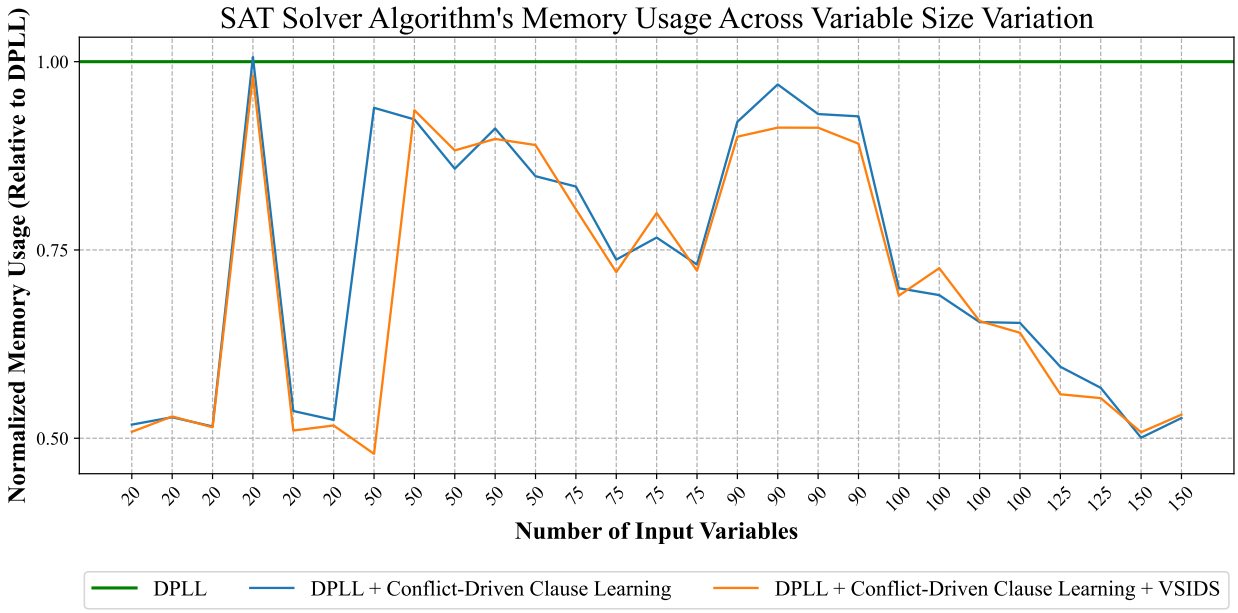


Figure 5: Memory Usage for Various Heuristics

Figure 5 campares memory usage between the basic DPLL algorithm (green line) and its enhanced versions incorporating Conflict-Driven Clause Learning (blue line) and VSIDS (orange line). Both enhancements consistently use less memory than the base DPLL, with normalized memory usage falling below 1.0 across most variable sizes. While CDCL and VSIDS introduce additional data structures, their more efficient search space pruning results in lower overall memory consumption. The DPLL + CDCL + VSIDS configuration shows slightly better memory efficiency than DPLL + CDCL alone, particularly for larger input sizes, indicating that the combined strategy not only improves runtime but also manages memory more effectively.

Thus, the two graphs collectively demonstrate that enhancing the DPLL algorithm with Conflict-Driven Clause Learning and VSIDS significantly improves solver efficiency. The time efficiency graph shows substantial reductions in computation time, especially for larger problem instances, while the memory usage graph indicates that these enhancements also lead to better memory management. Overall, the combined use of CDCL and VSIDS results in a more scalable and resource-efficient SAT solver across varying problem complexities.

# 6 Conclusion

In conclusion, we have successfully implemented a SAT solver based on DPLL, enhanced with Conflict-Driven Clause Learning and the VSIDS heuristic. Experimental results confirm correct functionality of our design and show that these enhancements significantly improve performance on complex instances.

Based on the results and analysis presented, it is evident that augmenting the classical DPLL algorithm with Conflict-Driven Clause Learning and the VSIDS heuristic substantially enhances the performance of SAT solvers. Our experiments reveal that these techniques not only reduce computation time but also optimize memory usage, particularly for complex problem instances with large variable and clause counts. The integration of CDCL enables the solver to avoid redundant search paths by learning from conflicts, while VSIDS guides the decision-making process more effectively by prioritizing impactful variables. Together, these improvements make the solver more robust, efficient, and suitable for solving real-world SAT problems.

The complete source code and implementation details are available at:
https://github.com/vishalkevat007/SAT-Solver-Implementation

# References


[1] João P Marques-Silva and Karem A Sakallah. "Boolean satisfiability in electronic design automation". In: *Proceedings of the 37th Annual Design Automation Conference*. 2000, pp. 675–680.

[2] Muhammad Osama and Anton Wijs. "Multiple decision making in conflict-driven clause learning". In: *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2020, pp. 161–169.

[3] Jia Hui Liang et al. "Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers". In: *Hardware and Software: Verification and Testing: 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings 11*. Springer. 2015, pp. 225–241.