ECE51216 Digital Systems Design Automation

Implementation of DPLL SAT Solver (Option 1) – Final Project Report

Team members: Harshini Jaiganesh, Kaarthikeya Karanam

Introduction

The Boolean Satisfiability Problem (SAT) is fundamental to many computational problems, where the goal is to determine if a given Boolean formula can be satisfied. Efficient SAT solvers enable automated reasoning, optimization, and decision-making, making them indispensable in applications such as circuit design, software verification, and combinatorial problem solving.

The DPLL SAT solver is a fundamental algorithm that systematically explores the search space using backtracking and unit propagation. It recursively assigns truth values to variables and prunes the search space through logical deductions. DPLL introduced key techniques like pure literal elimination and clause learning, significantly improving the efficiency of SAT solving. However, as problem sizes increased, more advanced heuristics became absolutely essential in order to handle complex instances.

Rationale

Improvising the DPLL with advanced heuristics like Conflict-Driven Clause Learning (CDCL) and Variable State Independent Decaying Sum (VSIDS) has led to significant improvements in the performance of the SAT solver. CDCL learns from conflicts to prune the search space more effectively, while VSIDS dynamically prioritizes variables based on conflict frequency. These optimizations help modern SAT solvers navigate large and intricate search spaces more efficiently, making them highly effective for a multitude of applications.

Instead of just reversing the most recent decision, CDCL also introduces non-chronological backtracking, which enables the solver to go back to the pertinent decision level that triggered the conflict. Assuming that variables that have recently been involved in conflicts are more important to the structure of the problem, VSIDS prioritizes them rather than selecting them at random or in a predetermined order. To make sure the solver adjusts to shifting p roblem regions, the activity score of each variable is incremented each time it appears in a learned clause. Over time, all scores progressively decrease. This method speeds up overall convergence and conflict resolution by assisting the solver in concentrating on the most "active" portions of the formula.

A feedback loop is created when CDCL and VSIDS work together. CDCL learns from conflicts to steer clear of unproductive paths, and VSIDS uses this conflict information to inform future variable selections. Modern SAT solvers can solve problems that would be impossible with older, less adaptive strategies thanks to this synergy, which makes them not just helpful but crucial.

Algorithm/Pseudo Code:

DPLL Algorithm

The **Davis-Putnam-Logemann-Loveland (DPLL)** algorithm is a foundational method for solving Boolean satisfiability (SAT) problems. Introduced in the 1960s, it uses a recursive backtracking approach to explore possible truth assignments for variables, forming the core of many contemporary SAT solvers.

The main components of the DPLL algorithm include:

- ***Backtracking Mechanism****:* DPLL explores the space of variable assignments by recursively trying truth values. If a conflict arises, it backtracks to explore alternative assignments, ensuring completeness in the search process.
- ***Unit Propagation****:* When a clause contains only one unassigned literal, that literal's value is logically forced. This propagation simplifies the formula and reduces the number of decisions needed.
- ***Pure Literal Elimination****:* If a literal appears exclusively in a single polarity (either always positive or always negative), it can be assigned a satisfying value without affecting the satisfiability of the formula. This optimization reduces complexity.

Although the original DPLL algorithm is relatively simple, it forms the backbone of modern SAT solvers. Enhancements such as **conflict-driven clause learning (CDCL)** and **watched literal schemes** have significantly boosted its efficiency, enabling it to solve large-scale SAT instances effectively.

Inputs:

- `cnf_formula`: A list containing all the clauses in conjunctive normal form.
- `set_of_literals`: A collection of all literals present in the CNF.

Output:

- `SAT` or `UNSAT`, depending on whether the formula can be satisfied.
- Assignment of literals that satisfy the formula.

Overview of the Algorithm:

- The algorithm starts with **unit propagation**, simplifying the formula by resolving single-literal clauses.
- It checks for **empty clauses** (signifying UNSAT) and for full satisfaction (all clauses resolved).
- It then selects the most frequently occurring literal among the unresolved clauses and makes a **decision**.
- Two branches are formed: one where the literal is assumed true, and another where it is assumed false.
- The CNF is reduced accordingly in each branch, and the algorithm **recursively** checks both.
- If both branches result in failure (UNSAT), the original formula is declared unsatisfiable.

Pseudo Code:

```
dpll(cnf_formula, set_of_literals):
    # Unit propagation
    for clause in cnf_formula:
        if length(clause) == 1:
            cnf_formula, set_of_literals = unit_propagation(cnf_formula,
                                                            set_of_literals)

    # Check if a clause is empty
    if [] in cnf_formula:
        return UNSAT

    # Check if all clauses are satisfied
    if not cnf_formula:
        return SAT

    # Initialize literal set
    s = set_of_literals

    # Choose the most common literal
    t = most_common(cnf_formula)

    # Make copies of CNF formula
    cnf_formula_copy = cnf_formula



    # Remove the chosen literal from the literal set
    if t in s:
        s.remove(t)

    # Get reduced CNF formula after setting the chosen literal to true
    r1 = reduced(cnf_formula, t)

    # Get reduced CNF formula after setting the chosen literal to false
    r2 = reduced(cnf_formula_copy, -t)

    # Recursive DPLL calls with reduced CNF formulas
    return dpll(r1, s) or dpll(r2, s)
```

Heuristics Used

a) Conflict-Driven Clause Learning (CDCL):

Conflict-Driven Clause Learning (CDCL) is a heuristic aimed at improving the performance of SAT solvers by learning from conflicts encountered during the search. When a conflict arises, meaning the current set of variable assignments contradicts one or more clauses, the algorithm analyzes the conflict to identify its root cause and creates a new clause that captures this information. This clause is then added to the problem, guiding the solver away from repeating the same erroneous paths. Unlike basic backtracking, CDCL allows for non-chronological backtracking, jumping directly to the decision level where the conflict originated. This significantly reduces the number of possibilities the solver must explore, enhancing its efficiency.

Inputs:

- cnf_formula: The SAT problem represented in DIMACS CNF format.
- heuristic: Specifies the method for selecting decision literals; values can be 0, 1, or 2.
- conflicts_limit: A threshold for the number of conflicts before triggering a restart.
- literal_block_dist_limit: A threshold for the Literal Block Distance (LBD), which influences clause learning.

Outputs:

- sat: A boolean result indicating whether the formula is satisfiable (True) or unsatisfiable (False).
- model: A list of integers representing the satisfying assignment if the formula is satisfiable, or an empty list otherwise.

Overview:

The CDCL algorithm is a robust method for solving Boolean SAT problems.

a) It starts with **unit propagation**, assigning values based on current constraints.

b) If not all variables can be determined, the solver uses **decision heuristics** to choose values for unassigned variables and explores the resulting implications.

c) When contradictions occur, the solver performs **conflict analysis**, learns a new clause to avoid repeating the same conflict, and adds it to the formula.

d) It then **backtracks**, often non-chronologically, to an earlier state that avoids the conflict.

e) This process continues iteratively until a solution is found or the solver concludes that no solution exists.

f) Through conflict-driven learning and smart decision strategies, CDCL effectively prunes the search space and enhances the SAT solver's ability to find solutions efficiently.

Pseudo Code:

Function CDCL(cnf_formula, heuristic, conflicts_limit, literal_block_dist_limit):

```
Initialize:
    decision_level ← 0
    decisions ← 0
    unit_propagations ← 0
    reinstates ← 0
    conflicts ← 0
```

```
# Initial unit propagation
(propagated_literals, conflict_clause) ← unit_propagation(decision_level)
unit_propagations ← unit_propagations + length(propagated_literals)
if conflict_clause is not None:
    return UNSAT, empty model, decisions, unit_propagations, reinstates

while not all variables assigned:

    # Choose decision literal based on heuristic
    decision_literal ← select_decision_literal(heuristic)
    if decision_literal is None:
        break

    decision_level ← decision_level + 1
    assign_literal(decision_literal, decision_level)
    decisions ← decisions + 1

    # Propagate implications of decision
    (propagated_literals, conflict_clause) ← unit_propagation(decision_level)
    unit_propagations ← unit_propagations + length(propagated_literals)

    while conflict_clause is not None:

        conflicts ← conflicts + 1

        if conflicts ≥ conflicts_limit:
            conflicts ← 0
            conflicts_limit ← conflicts_limit × 1.1
            literal_block_dist_limit ← literal_block_dist_limit × 1.1
            reinstates ← reinstates + 1
            decision_level ← 0
            reinstate_formula()
            delete_learned_clauses_by_lbd(literal_block_dist_limit)
            break

        backtrack_level ← analyze_conflict(conflict_clause, decision_level)

        if backtrack_level < 0:
            return UNSAT, empty model, decisions, unit_propagations, reinstates
        backtrack_to_level(backtrack_level)
        decision_level ← backtrack_level

        (propagated_literals, conflict_clause) ← unit_propagation(decision_level)
        unit_propagations ← unit_propagations + length(propagated_literals)

return SAT, current_model, decisions, unit_propagations, reinstates
```

b) Watched Literals:

The watched literals technique is a heuristic used to efficiently manage clause evaluation in Boolean formulas. Instead of monitoring all literals in a clause, the solver selects one or two key literals to "watch" for changes in their truth values. When a watched literal is updated, the solver re-examines the clause to check whether it remains satisfied or whether further action is needed, such as unit propagation or conflict handling. This strategy significantly reduces the number of clause checks required, enhancing solver performance.

Inputs:

- `assignment`: A list representing the current values assigned to variables.
- `new_variable`: The most recently assigned variable, used to trigger updates in watched literals.

Outputs:

- `success`: A boolean indicating whether the clause remains valid after the update (`True`) or has become unsatisfied (`False`).
- watched_one and watched_two get updated when the newly assigned variable matches the current watched_two, prompting a swap to maintain consistency. Additionally, watched_one is updated during propagation if its current literal becomes false and an alternative unwatched, non-false literal is found.

Function Behavior:

a) If the newly assigned variable corresponds to the second watched literal, the algorithm swaps the two watched literals to preserve consistent processing.
b) It then checks the clause's status: whether it's satisfied, unsatisfied, or needs to update one of its watched literals.
c) If any watched literal evaluates to `True`, the clause is satisfied, and no further action is needed.
d) If both watched literals are `False`, the clause is unsatisfied, and the update fails.
e) If one watched literal becomes `False` while the other is unassigned, the algorithm looks for another unassigned or non-false literal to replace the failed one—ensuring it isn't the same as the second watched literal. If found, the clause continues in an unresolved state or becomes a unit clause if only one literal remains unassigned.

Pseudo Code:

```
Function watched_literal_update_pass(assignment, new_variable)

1. If new_variable is the second watched literal:
    a. Swap watched_one and watched_two
2. If clause is satisfied due to current assignment:
    a. Return (True, watched_one_literal, False)
3. If both watched literals are assigned False:
    a. Return (False, watched_one_literal, False) ⟶ conflict detected
4. If watched_one is False and watched_two is unassigned:
    a. Try to find a new literal to watch (other than watched_two), that is not currently False
    b. If no such literal found:
        i. Clause is unit
       ii. Return (True, watched_two_literal, True)
    c. Else:
        i. Update watched_one to new index
       ii. Return (True, watched_one_literal, False)
```

c) Variable State Independent Decay Sum (VSIDS):

VSIDS is a branching heuristic commonly used in SAT solvers to determine which variable to branch on next during the search process. It assigns dynamic scores to variables based on how frequently they appear in recent conflicts. Variables that have been involved in more recent conflicts are given higher priority, as they are more likely to influence the solver's progress. To keep the focus on current conflicts, the heuristic includes a decay mechanism that gradually reduces the influence of older conflicts. This ensures that the solver remains responsive to recent developments, helping it make better-informed decisions as it explores possible solutions.

VSIDS_heuristic :

Inputs:

- variables: A list of all variables in the formula.
- assignment: A dictionary showing the current assignments of the variables.
- positive_literal_counter: Tracks how often positive literals appear in learned clauses.
- negative_literal_counter: Tracks how often negative literals appear in learned clauses.

Output:

- decision_literal: The selected literal for branching, chosen based on the highest frequency of appearance in learned clauses.

Description:

This function applies the VSIDS heuristic to guide variable selection in a SAT solver. It scans through the list of unassigned variables, checking the occurrence of both their positive and negative literals in learned clauses. The literal with the highest frequency is chosen as the next branching decision. This approach favors variables that have frequently contributed to conflicts, under the assumption that resolving these will lead to faster convergence toward a solution.

<u>Pseudo Code:</u>

```
vsids_heuristic():

    decision_literal = None

    best_counter = 0

    for variable in variables:

        if assignment[variable] == 0:

            if positive_literal_counter[variable] > best_counter:

                decision_literal = variable

                best_counter = positive_literal_counter[variable]



            if negative_literal_counter[variable] >= best_counter:

                decision_literal = -variable

                best_counter = negative_literal_counter[variable]

    return decision_literal
```

Description of Data structures and Programming Constructs:

Classes & OOPS structures:

1. *Clause_CNF - Class*

- **Efficiency**: The `Clause_CNF` class provides a compact and efficient way to represent individual clauses by storing their literals and associated metadata. This design allows for fast access and manipulation of literals during propagation and conflict analysis.
- **Modularity**: Encapsulating clause-related functionality in its own class promotes modularity, making it easy to update or extend the representation of clauses without affecting other parts of the solver.
- **Clarity**: Keeping clause operations within a dedicated class improves code readability and maintainability, allowing developers to focus on clause-specific behavior in a clean and organized manner.

2. *CNF_Formula - Class*

- **Organization**: The `CNFFormula` class structures the overall representation of the problem by managing clauses, variables, and other relevant components. This helps keep the logic organized and easier to navigate.
- **Abstraction**: By abstracting the CNF formula as a single class, high-level operations such as clause addition, simplification, and traversal can be performed without delving into the low-level details of its internal representation.
- **Encapsulation**: The formula's internal state and operations are well-encapsulated, minimizing side effects and making the codebase easier to debug and test.

Data Structures:

In the SAT solver implementation, various data structures are fundamental to efficiently managing elements of the solving process such as clauses, variable assignments, watched literals, and decision-making. Below is a breakdown and rationale for the key data structures used:

LIST
Usage: formula

- **Structure:** A list of lists where each inner list represents a clause consisting of literals.
- **Purpose:** Stores the input Boolean formula in a structured form for the solver to process.
- **Reasoning:** Lists make it easy to iterate through and modify clauses and their literals during solving.

Usage: clauses

- **Structure:** A list of Clause objects.
- **Purpose:** Maintains the formula as a collection of clause instances for organized access.
- **Reasoning:** Encapsulating clauses as objects supports better maintainability and enables clause-level operations.

Usage: assignment, antecedent, decision_level

- **Structure:** Lists indexed by variable, storing current assignment, reason for assignment (antecedent), and decision level.
- **Purpose:** These lists manage key information about variable states during solving.
- **Reasoning:** Lists allow constant-time indexing and updates, making them suitable for dynamic variable tracking as the formula evolves.

SET
Usage: variables

- **Structure:** A set of all distinct variables used in the formula.
- **Purpose:** Maintains a collection of unique variables.
- **Reasoning:** Sets automatically eliminate duplicates and offer efficient membership checks and set operations.

STACK
Usage: assignment stack, Decision levels

- **Structure:** Layered structure that tracks variable assignments, their causes, and decision depths, enabling efficient backtracking and conflict resolution.
- **Purpose:** Prioritizes the backtracking as they allow us to revoke the previous assignments.
- **Reasoning:** Simplify state management, support efficient backtracking and conflict resolution, and ensure the solver remains logically consistent throughout the search process.

DICTIONARY
Usage: watched_lists

- **Structure:** Maps each literal to a list of clauses in which it's being watched.
- **Purpose:** Supports the watched literal scheme for fast propagation.
- **Reasoning:** Enables quick retrieval and updates of clauses by literal, essential for unit propagation and conflict resolution.

TUPLE
Usage: Functions like cdcl()

- **Structure:** Tuples are immutable sequences used to group multiple return values from a function.
- **Purpose:** They allow functions such as cdcl() to return multiple pieces of information simultaneously in a compact form.
- **Reasoning:** Tuples are lightweight and immutable, making them ideal for scenarios where grouped data needs to be returned without the overhead or mutability of more complex data types like lists or dictionaries. Their use enhances clarity and performance when returning structured outputs.

NUMPY-ARRAYS
Usage: Tracking literal activity for VSIDS heuristic

- **Structure:** NumPy arrays initialized with zeros to represent activity scores for literals.
- **Purpose:** They maintain and update the activity levels of literals to support the Variable State Independent Decaying Sum (VSIDS) decision heuristic.
- **Reasoning:** NumPy arrays offer efficient numerical operations and better performance compared to native Python lists. Their fixed-size, contiguous memory layout allows for rapid updates and computations, which is crucial for performance-sensitive components like heuristic scoring in SAT solvers.

Functions:

- ***partial_assignment_literals:*** This function returns the unassigned literals from a clause if none of its literals are already satisfied by the current assignment. If any literal in the clause is satisfied, it returns an empty list immediately.

- ***watched_literal_update_pass:*** This function updates watched literals in a clause when a new variable is assigned, checking whether the clause is satisfied, unsatisfied, or becomes a unit clause. It returns a tuple indicating the clause status, the key literal involved, and whether unit propagation is needed.
- ***is_satisfied:*** This function checks if the clause is satisfied under the current assignment using the two watched literals. It returns "True" if watched literal is assigned in a way that makes the clause true.
- ***literal_assignment:*** This function assigns a literal at a given decision level and updates all affected watched literals accordingly. It detects conflicts or new unit clauses, queues them for propagation, and returns whether the assignment was successful.
- ***all_assigned:*** This function checks whether all variables in the problem have been assigned a value. It returns `True` if the number of assigned variables matches the total number of variables.
- backtrack: This function undoes variable assignments made after a specified decision level by popping from the assignment stack. It resets the assignment, decision level, and previous value for each backtracked variable.
- ***conflict_inspection:*** This function performs conflict analysis using clauses learning to identify an assertive clause when a conflict occurs during SAT solving. It resolves conflicting clauses, updates activity scores, computes the assertion level and LBD (Literal Block Distance), and returns the new backtrack level for non-chronological backtracking.
- ***unit_propagation:*** The "unit_propagation" function propagates unit clauses by assigning their literals and checking for conflicts, returning either all propagated literals or a conflicting clause.
- ***vsids_heuristic:*** The "vsids_heuristic" function selects the next decision literal based on VSIDS, prioritizing the unassigned variable with the highest activity score.
- ***most_recurring_heuristic:*** This function selects the unassigned literal that appears most frequently in unsatisfied clauses using a simple frequency-based heuristic. It returns to the literal (positive or negative) that maximizes clause occurrences for potential decision-making.
- ***random_heuristic:*** This function randomly selects an unassigned variable, assigns it a random polarity (positive or negative), and removes it from the internal unassigned list. It ensures randomness in decision-making for SAT solvers when no specific heuristic is applied.
- ***select_decision_literal:*** This function selects a decision literal based on the specified heuristic (VSIDS, most recurring, or random). It raises an error if an unknown heuristic value is provided.
- ***cdcl:*** The "cdcl" function implements a conflict-driven clause learning (CDCL) algorithm to solve a CNF formula by making decisions, propagating units, and handling conflicts with backtracking and learned clauses. It tracks decisions, unit propagations, and reinstates while managing conflict limits and literal block distance for clause deletion.
- ***delete_learned_clauses_by_lbd:*** This function removes learned clauses with a literal block distance (LBD) greater than a specified limit from the watched lists. It retains clauses with lower LBD in the learned clauses list and updates the watched lists accordingly.
- ***find_model:*** The "find_model" function reads a CNF formula from a DIMACS file, applies the CDCL algorithm to solve it, and returns the result, including whether the formula is satisfiable, the model (variable assignments), CPU time, and statistics on decisions, unit propagations, and reinstates. It handles file reading errors and provides a formatted output.

Benchmarks and Performance Analysis:

The SAT solver is evaluated using a systematic process composed of the following stages:

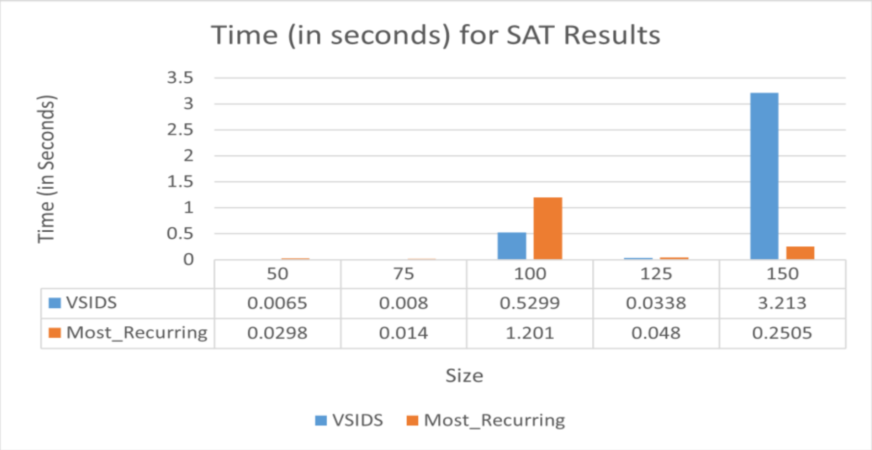
1. **Initialization**: The solver receives input in the form of a CNF file formatted according to the DIMACS standard. These files include comment lines (prefixed by 'c'), a problem line (starting with 'p') that declares the number of variables and clauses, followed by the clauses themselves—each ending with a `0`. If the CNF formula is satisfiable, the solver outputs a valid variable assignment both to the terminal and to a file in the solutions directory; otherwise, it reports the formula as unsatisfiable.
2. **CNF File Parsing**: The ***find_model*** function processes the input CNF file, extracts clause data, and returns a list of clauses (where each clause is a list of literals), along with the total number of variables and clauses.
3. **SAT Solving Workflow**: The solver employs the Conflict-Driven Clause Learning (CDCL) algorithm, consisting of the following key components,
    - **Decision Heuristics**: The solver strategically selects unassigned variables using heuristics like VSIDS or Most Recurring to guide the search toward conflicts faster.
    - **Unit Propagation**: After each decision, the solver repeatedly applies the unit clause rule to simplify the formula and infer necessary assignments.
    - **Conflict Detection and Analysis**: When a conflict is encountered (i.e., a clause is unsatisfiable under current assignments), the solver analyzes the conflict to identify the root cause.
    - **Clause Learning**: From the conflict, a new clause (called a learned clause) is generated and added to the formula to prevent the solver from repeating the same mistake.
    - **Backtracking**: Rather than undoing decisions step-by-step, the solver uses conflict information to jump back to the most relevant earlier decision level, improving efficiency.
    - **Clause Deletion**: To manage memory and performance, the solver periodically deletes learned clauses with low relevance (e.g., based on Literal Block Distance, or LBD).
    - **Restarts**: If progress stagnates, the solver may reinstate to a clean state while retaining learned clauses, allowing it to explore new parts of the search space more effectively.
4. **Performance Evaluation:** The solver's effectiveness is measured using several metrics, including time to solve, number of learned clauses, decisions made, and implications derived. These indicators help in assessing the solver's computational efficiency and accuracy.

Benchmarks for evaluation:

We used the SATLIB benchmarks to evaluate the performance of our SAT solver on both SAT and UNSAT problems. Each file is SATLIB Benchmark problems of varying size of literals ranging from 20 to 200 and clauses from 91 to 860 corresponding to number of literals with both SAT and UNSAT. Benchmark uf20-01.cnf is copied to same folder as python file to execute directly.

Observations:

For SAT problems of sizes 20, 50, 75, 100, and 125, the first few instances. Typically, between 5 and 10 were selected for experimental evaluation. The following figures display the average CPU time, number of decisions, unit propagations, and restarts observed across these samples.



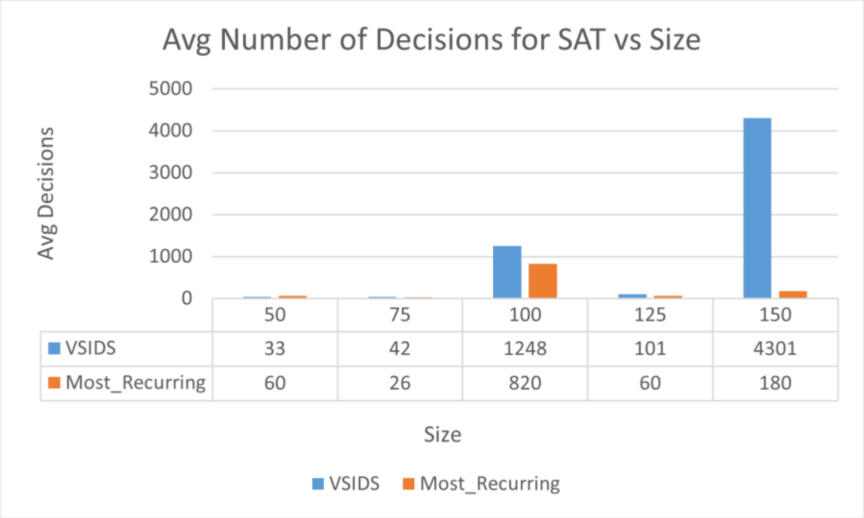
	50	75	100	125	150
🟦 VSIDS	0.0065	0.008	0.5299	0.0338	3.213
🟧 Most_Recurring	0.0298	0.014	1.201	0.048	0.2505

Fig1: Comparison of SAT example Size vs Time (in Seconds)

For smaller sizes (50 to 75): Both VSIDS and Most_Recurring have relatively small times, with VSIDS being noticeably faster than Most_Recurring.

For medium sizes (100 to 125): The time for Most_Recurring increases substantially, especially at size 100, where it takes significantly more time than VSIDS.

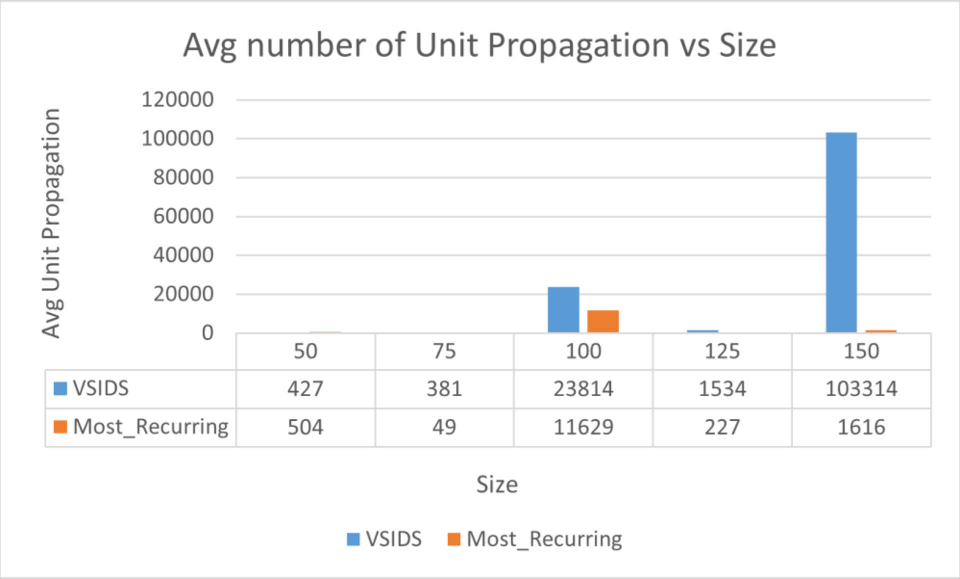
For larger sizes (150): The time for Most_Recurring stays relatively low (compared to size 100), but still, VSIDS remains more efficient in terms of time. VSIDS increases sharply for larger problem sizes, particularly at size 150, but it does not match the exponential jump seen in the Most_Recurring method at size 100.



	50	75	100	125	150
🟦 VSIDS	33	42	1248	101	4301
🟧 Most_Recurring	60	26	820	60	180

Fig2: Comparison of SAT Example size Vs Number of decisions

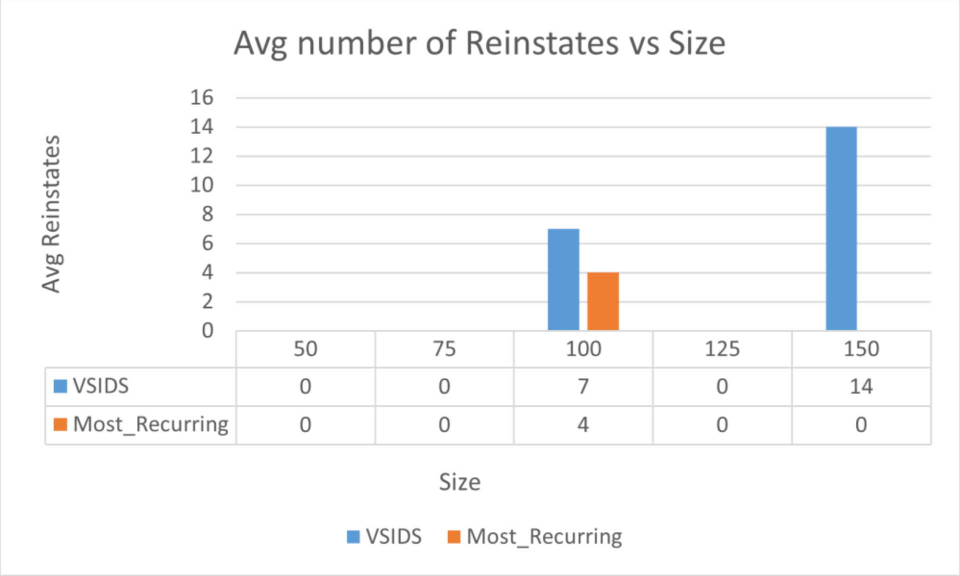
This bar graph compares the average number of decisions made by two heuristics—**VSIDS** and **Most_Recurring**—for solving SAT problems of increasing size. While both perform similarly for smaller sizes, **VSIDS shows significantly higher decision counts at larger sizes (100 and 150)**, indicating it may be less efficient than Most_Recurring as complexity grows.



	50	75	100	125	150
■ VSIDS	427	381	23814	1534	103314
■ Most_Recurring	504	49	11629	227	1616

Fig3: Comparison of SAT Example size Vs Number of Unit propagations

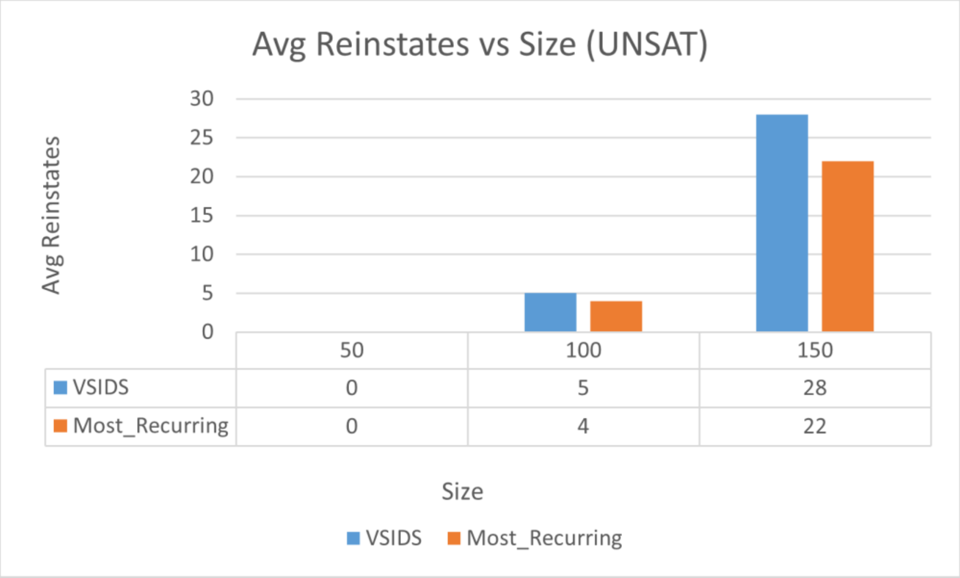
The chart shows that Most_Recurring consistently requires fewer unit propagations than VSIDS, especially as SAT problem size increases. While both perform similarly at smaller sizes, VSIDS shows exponential growth in propagations (e.g., over 100,000 at size 150), whereas Most_Recurring remains relatively stable (just over 1,600). This suggests that Most_Recurring scales more efficiently and is better suited for larger SAT instances.



	50	75	100	125	150
■ VSIDS	0	0	7	0	14
■ Most_Recurring	0	0	4	0	0

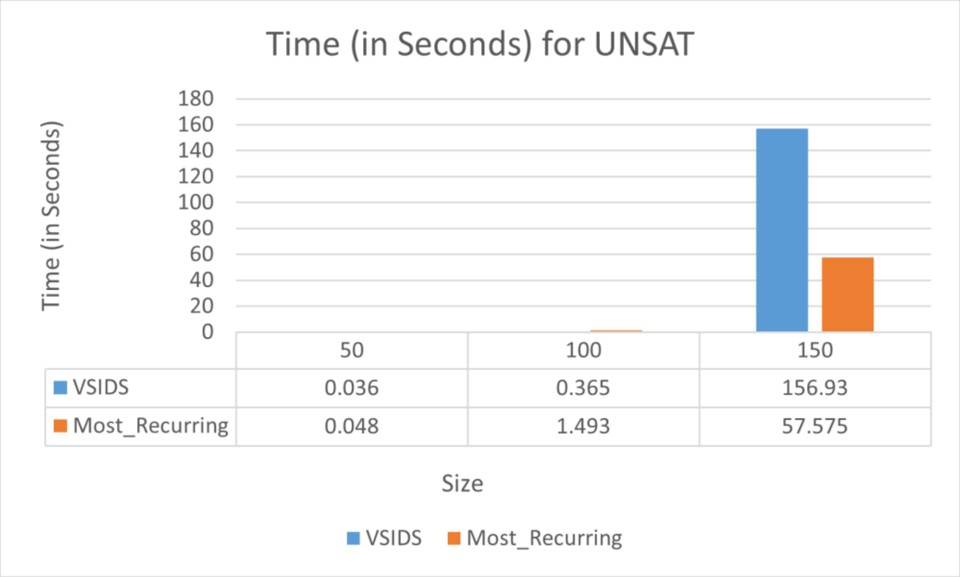
Fig4: Comparison of SAT Example size Vs Number of restarts

Restarts are infrequent in small SAT problems, but their occurrence slightly increases with problem size. The order of restart frequency is: Most Frequent < VSIDS, a trend that also holds true for UNSAT cases. Notably, both Most Frequent and VSIDS show minimal variation in restart count across SAT and UNSAT instances.



	50	100	150
🟦 VSIDS	0	5	28
🟧 Most_Recurring	0	4	22

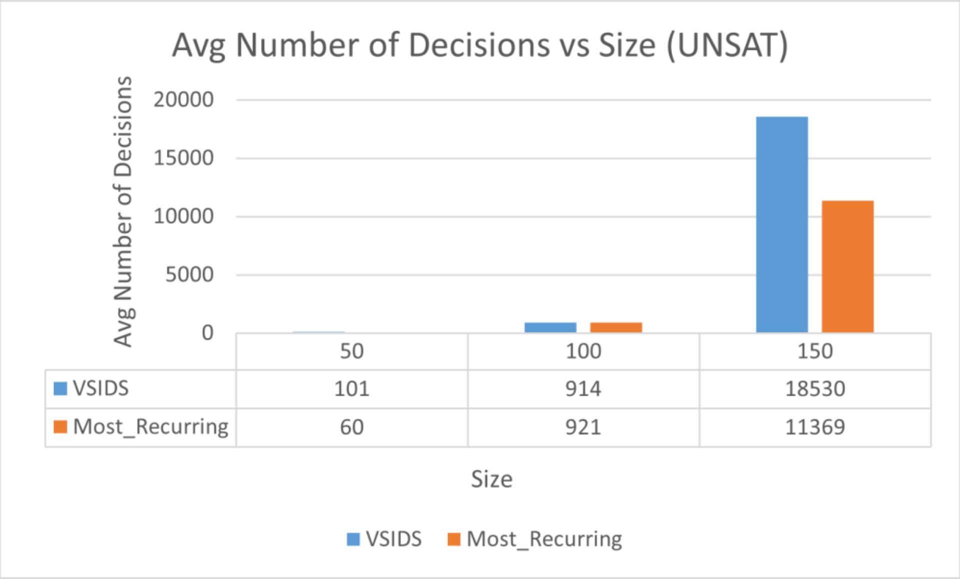
Fig5: Comparison of UNSAT example size Vs Number of restarts – UNSAT



	50	100	150
🟦 VSIDS	0.036	0.365	156.93
🟧 Most_Recurring	0.048	1.493	57.575

Fig6: Comparison of UNSAT example size Vs Time (in Seconds)

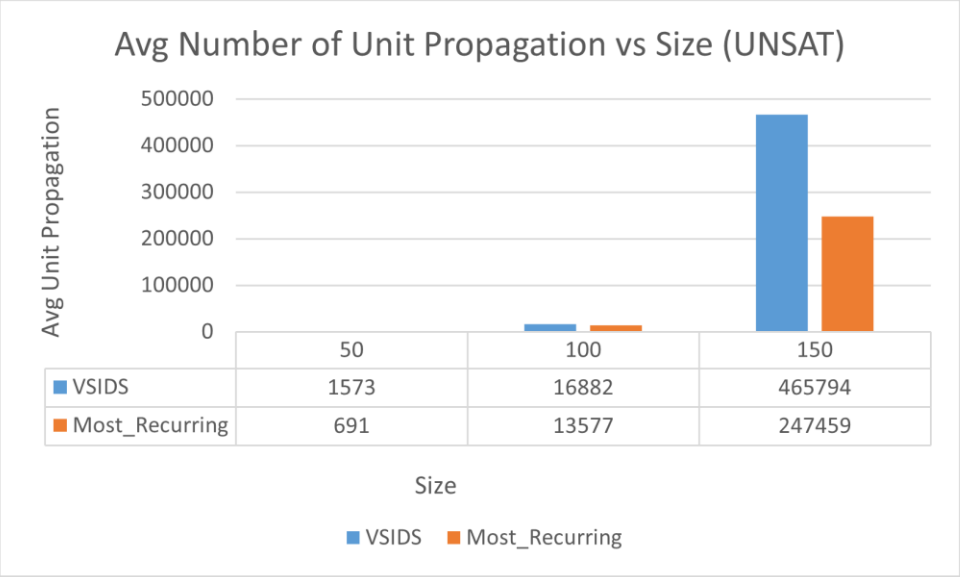
As the UNSAT problem size increases, Most_Recurring consistently outperforms VSIDS in solving time. While VSIDS is slightly faster at size 50 and 100, it becomes significantly slower at size 150 (156.93s vs. 57.575s). This highlights that Most_Recurring scales more efficiently, making it a better choice for large UNSAT instances.



	50	100	150
VSIDS	101	914	18530
Most_Recurring	60	921	11369

Fig7: Comparison of UNSAT example Size vs Number of Decisions

This chart shows the average number of decisions needed by VSIDS and Most_Recurring heuristics for UNSAT problems as the problem size increases. Both heuristics perform similarly at smaller sizes, but at size 150, Most_Recurring significantly outperforms VSIDS, requiring about 6,000 fewer decisions, indicating better efficiency in larger, unsatisfiable cases.



	50	100	150
VSIDS	1573	16882	465794
Most_Recurring	691	13577	247459

Fig8: Comparison of UNSAT example Size vs Number of Unit Propagations

This graph compares the average number of unit propagations required by VSIDS and Most_Recurring heuristics for UNSAT problems across different sizes. As problem size increases, unit propagations rise sharply, especially for VSIDS. At size 150, Most_Recurring achieves nearly half the propagations of VSIDS (247k vs. 466k), indicating that it is significantly more efficient in pruning the search space for larger unsatisfiable instances.

KEY TAKEAWAYS:

For benchmark files of size 20, the average execution time remains nearly constant across all heuristics. However, as the problem size increases, VSIDS consistently performs faster on average, followed by Most Frequent. This trend is also observed in UNSAT problems of sizes 50, 75, and 100.

When evaluating the average number of decisions made across SAT and UNSAT benchmarks, the Most Frequent heuristic requires the fewest decisions, followed by VSIDS. In terms of unit propagations, VSIDS results in the highest average number across all benchmark sizes. For UNSAT problems specifically, Most Frequent yields the lowest number of unit propagations. Overall, CDCL enhanced with heuristics outperforms the basic DPLL algorithm across all metrics: decisions, unit propagations, and execution time.

The results clearly show that the Conflict-Driven Clause Learning (CDCL) solver outperforms the traditional DPLL solver with watched literals in all metrics, including runtime, number of decisions, unit propagations, and restarts. This highlights the significance of incorporating advanced methods such as conflict clause learning and non-chronological backtracking to improve SAT solving efficiency.

Conclusion:

The implementation of the DPLL SAT Solver with CDCL and VSIDS heuristics enhances traditional SAT solving by improving efficiency and decision-making. By integrating conflict-driven learning and dynamic variable prioritization, the solver effectively handles complex SAT instances. Performance evaluation will demonstrate the impact of these heuristics on search space reduction and execution time. The project concludes with a fully functional solver, detailed documentation, and a comprehensive report analyzing results and optimizations.

References

[1] L. Zhang and S. Malik, “The Quest for Efficient Boolean Satisfiability Solvers”, CAV 2002

[2] https://www.mqasem.net/sat/sat/

[3] https://people.eecs.berkeley.edu/~sseshia/219c/lectures/SATSolving.pdf

[4] https://www.cs.princeton.edu/~zkincaid/courses/fall18/readings/SATHandbook-CDCL.pdf

[5] https://cdn.aaai.org/ojs/12164/12164-13-15692-1-2-20201228.pdf