

Project-1: Implement CDCL with two watched literals

Team 8: Abhishek Shastry | Veetrag Nahar

Overview

This code is a Python implementation of the Conflict-Driven Clause Learning (CDCL) algorithm for solving Boolean satisfiability (SAT) problems represented in the DIMACS CNF format. It incorporates various features of modern SAT solvers, such as unit propagation using watched literals, clause learning, non-chronological backtracking, an arithmetic restart policy, and the Variable State Independent Decaying Sum (VSIDS) heuristic for variable selection. Taking inspiration from CDCL-SAT-Solver [1], we have implemented the arithmetic restart policy and the VSIDS heuristic ourselves in the given code. The arithmetic restart policy periodically restarts the solver's search process after a certain number of conflicts, following an arithmetic progression, to escape unpromising areas of the search space. The VSIDS heuristic prioritizes variables involved in recent conflicts for branching decisions by maintaining activity scores that decay over time, guiding the search towards solutions more effectively. Here is a detailed description:

1. Importing Libraries and Modules:

- The code starts by importing various Python libraries and modules that are necessary for its functionality. These include **sys** for handling command-line arguments, **random** for random operations, **time** for measuring execution time, **math** for mathematical functions, **textwrap** for formatting output, **pprint** for pretty-printing, **dataclasses** for defining data classes, **typing** for type hints, and **defaultdict** from **collections** for efficient dictionary handling.

2. Representing Literals, Clauses, and Formulas:

- The **Literal** class represents a Boolean literal, which is either a propositional variable or its negation. It has two attributes: **variable** (an integer representing the variable) and **negation** (a Boolean indicating whether the literal is negated or not). It provides methods for string representation and negation.
- The **Clause** class represents a disjunction (OR) of literals, which is a clause in the Conjunctive Normal Form (CNF). It stores a list of **Literal** objects and provides methods for string representation, iteration, length calculation, and hashing.
- The **Formula** class represents a conjunction (AND) of clauses, which is a CNF formula. It stores a list of **Clause** objects and keeps track of the set of variables present in the formula. During initialization, it removes duplicate literals from each clause and caches the set of variables. It provides methods for accessing variables, string representation, iteration, and length calculation.

3. Representing Assignments:

- The **Assignment** class represents an assignment of a Boolean value (True or False) to a variable at a particular decision level. It also stores an optional antecedent clause that caused the assignment.
- The **Assignments** class is a dictionary that stores the current assignments of variables. It keeps track of the current decision level and maintains variable activities for the Variable State Independent Decaying Sum (VSIDS) heuristic. It provides methods for incrementing variable activity, retrieving the value of a literal, assigning and unassigning variables, and checking if the assignments satisfy the formula.

4. Utility Functions:

- ***init_watches***: This function initializes the watched literal data structures (**lit2clauses** and **clause2lits**), which are used for efficient unit propagation. It creates mappings between literals and clauses, and clauses and their watched literals.
- ***cdcl_solve***: This is the main function that implements the Conflict-Driven Clause Learning (CDCL) algorithm. It initializes the necessary data structures, performs unit propagation on unit clauses, assigns values to variables based on the VSIDS heuristic, handles conflicts and clause learning, and backtracks, as necessary. It terminates after 90 seconds or when a satisfying assignment is found, or the formula is proven unsatisfiable.
- ***learntClauseAdd***: This function adds a learnt clause to the formula and updates the watched literal data structures.
- ***all_variables_assigned***: This function checks if all variables in the formula have been assigned a value.
- ***branchingVariablePick***: This function implements the VSIDS heuristic for selecting the next variable to assign a value (the Decide rule). It computes variable activities based on their involvement in conflicts and selects the variable with the highest activity score.
- ***backtrack***: This function backtracks the assignments to a specific decision level (the Backjump rule) by removing assignments of variables with a decision level higher than the backtrack level.
- ***clause_status***: This function determines the status of a clause (satisfied, unsatisfied, unit, or unresolved) based on the current assignments.
- ***unitpropagation***: This function performs unit propagation using the watched literal data structures (the Propagate rule). It updates assignments based on unit clauses and handles conflicts.
- ***resolve***: This function resolves two clauses on a given variable to produce a new clause, removing the variable from the resulting clause (used in conflict analysis).
- ***conflict_analysis***: This function analyzes a conflicting clause to determine the backtrack level and generate a learnt clause (the Explain and Learn rules).
- ***parse_dimacs_cnf***: This function parses a DIMACS CNF file and constructs the corresponding Formula object.

5. Main Execution:

- The code checks if the correct number of command-line arguments is provided (a DIMACS CNF filename).
- It reads the contents of the **DIMACS CNF** file from the provided filename.
- It parses the file contents and constructs the corresponding **Formula** object using **parse_dimacs_cnf**.
- It calls the **cdcl_solve** function to solve the SAT problem for the constructed formula.
- If a **satisfying assignment** is found, it prints the execution time and the assignments.
- If the formula is **unsatisfiable**, it prints the execution time and "Formula is UNSAT."

Algorithm: Enhanced CDCL SAT Solver

Step 1: Initialization

- Parse Input: Read a DIMACS CNF file and initialize a Formula object. This object contains Clauses made up of Literals.
- Setup Data Structures: Initialize necessary data structures:
- Assignments to store current variable values and decision levels.
- lit2clauses and clause2lits for efficient implementation of unit propagation using watched literals.

Step 2: Setup Watched Literals

- Initialize Watches: For each clause, set up watched literals that will help in efficient propagation:
- For unit clauses, watch only literal.
- For longer clauses, watch any two literals.

Step 3: Main Solving Loop

- Check for Initial Unit Clauses: Assign values to literals in unit clauses and propagate consequences.
- Repeat until Solution or Timeout:
- Conflict Detection: If a conflict is detected during propagation, perform conflict analysis.
- Backtracking: Based on the conflict analysis, backtrack to a safe decision level and update the assignments.
- Variable Selection (VSIDS): Select the next variable to assign using the VSIDS heuristic which prioritizes variables based on their involvement in recent conflicts.
- Decision Making: Assign a value to the selected variable and propagate the effects.
- Arithmetic Restart: If the number of conflicts since the last restart meets the threshold determined by an arithmetic sequence, reset all assignments, and restart the solving process.

Step 4: Conflict Analysis

- Analyze the Conflict: Identify the clause that caused the conflict and use it to determine the backtrack level and learn a new clause.
- Learn New Clauses: Add the learned clause to the formula to prevent future similar conflicts.

Step 5: Unit Propagation

- Propagate Effects: After each assignment, propagate its effects throughout the formula. Adjust watched literals as necessary to maintain efficient propagation.
- Handle Conflicts: If a conflict arises during propagation, trigger the conflict analysis and backtracking process.

Step 6: Check for Solution

- Verify Completion: Check if all variables are assigned without conflicts.
- Verify Formula Satisfaction: Ensure that all clauses are satisfied with the current assignments.

Step 7: Termination

- SAT Found: If all clauses are satisfied, print the assignments and declare the formula satisfiable (SAT).
- UNSAT Found: If no assignments satisfy the formula, declare it unsatisfiable (UNSAT).
- Timeout: If the solver runs longer than the specified timeout (e.g., 90 seconds), terminate the process and report the timeout.

```
Procedure CDCL-Solve(Formula)
  Initialize data structures (assignments, watches, activities)
  Initialize conflict count and restart parameters
  Parse DIMACS CNF input to extract variables and clauses

  while True:
    if all variables assigned:
      if formula is satisfied with current assignments:
        print "s SATISFIABLE"
        output assignments
        return
      else:
        print "s UNSATISFIABLE"
        return

    UnitPropagate()

    if conflict detected:
      reason, learnt_clause = AnalyzeConflict(assignments)
      if reason == UNSAT:
        print "s UNSATISFIABLE"
        return

      LearnClause(learnt_clause, assignments)

      Backtrack(to level specified in conflict analysis)

      UpdateRestartPolicy(conflict count)

    var, value = PickBranchingVariable(Formula, assignments)
    MakeDecision(var, value)
    IncrementDecisionLevel()

Procedure UnitPropagate()
  while propagations possible:
    if clause becomes unit under current assignments:
      Propagate the necessary value to satisfy the clause
    if clause is conflicting:
      raise conflict detected

Function AnalyzeConflict(assignments)
  Based on the current conflict, derive the conflicting clause
  Use resolution to deduce a new learnt clause
  Calculate backtrack level from conflict history
  return backtrack level, learnt_clause

Procedure LearnClause(learnt_clause, assignments)
  Add the learnt clause to the formula
  Update variable activities as per VSIDS
  Update watches for the new clause

Procedure Backtrack(level)
  Revert decisions and assignments up to the given level
  Adjust decision level

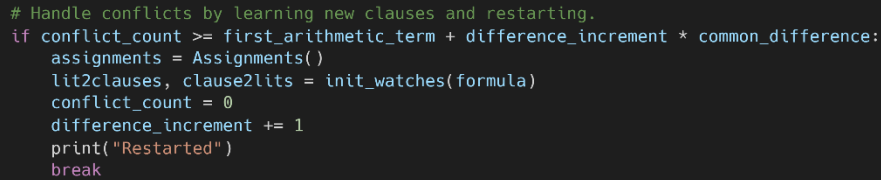
Function PickBranchingVariable(Formula, assignments)

Procedure Initialize(Formula)
  Setup initial variable activities, watches for each clause
  Determine initial decision level and prepare data structures for backtracking
```

Figure 1: High-level pseudocode overview of the CDCL SAT solver algorithm

Optimizations:

A) Arithmetic Restart policy

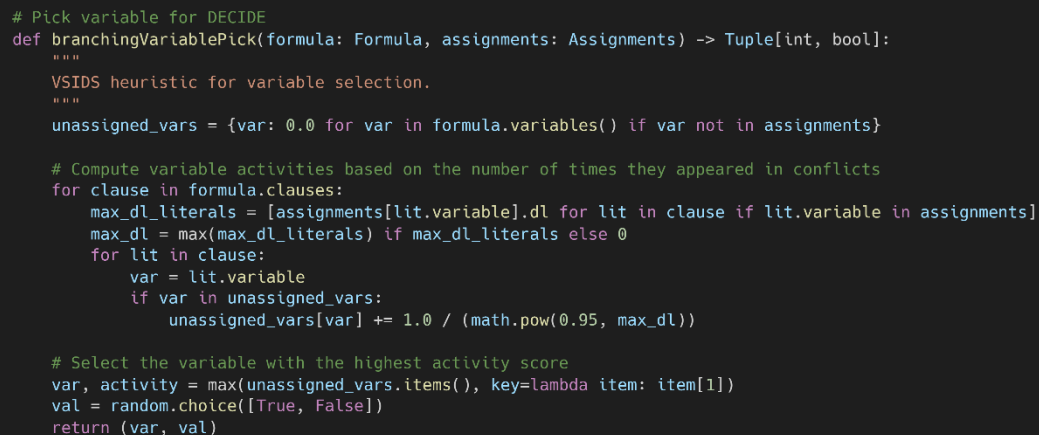


```
# Handle conflicts by learning new clauses and restarting.
if conflict_count >= first_arithmetic_term + difference_increment * common_difference:
    assignments = Assignments()
    lit2clauses, clause2lits = init_watches(formula)
    conflict_count = 0
    difference_increment += 1
    print("Restarted")
    break
```

Figure 2: Implementation of Arithmetic Restart policy

The arithmetic restart policy is a dynamic restart strategy used in SAT solvers and constraint satisfaction algorithms. It aims to strike a balance between exploring the search space thoroughly and avoiding getting stuck in unfruitful regions. The policy works by setting an initial restart threshold *first_arithmetic_term* and incrementing this threshold after each restart by a constant value *common_difference*. This increment is added to the previous increment *difference_increment*, forming an arithmetic progression for the restart thresholds. The idea is to allow the solver to explore the search space more thoroughly as the search progresses, while still periodically restarting to escape from potential local minima. The arithmetic restart policy has been shown to improve the performance of SAT solvers.

B) Variable State Independent Decaying Sum (VSIDS) heuristic



```
# Pick variable for DECIDE
def branchingVariablePick(formula: Formula, assignments: Assignments) -> Tuple[int, bool]:
    """
    VSIDS heuristic for variable selection.
    """
    unassigned_vars = {var: 0.0 for var in formula.variables() if var not in assignments}

    # Compute variable activities based on the number of times they appeared in conflicts
    for clause in formula.clauses:
        max_dl_literals = [assignments[lit.variable].dl for lit in clause if lit.variable in assignments]
        max_dl = max(max_dl_literals) if max_dl_literals else 0
        for lit in clause:
            var = lit.variable
            if var in unassigned_vars:
                unassigned_vars[var] += 1.0 / (math.pow(0.95, max_dl))

    # Select the variable with the highest activity score
    var, activity = max(unassigned_vars.items(), key=lambda item: item[1])
    val = random.choice([True, False])
    return (var, val)
```

Figure 3: VSIDS implemented under *branchingVariablePick*

The VSIDS heuristic is a popular method used in modern SAT solvers to decide which variable to branch on next during the search process. The main idea behind VSIDS is to give higher priority to the variables that have been involved in recent conflicts, as these variables are more likely to be part of the solution.

In *Figure 3*, the function *branchingVariablePick* takes two arguments: *formula*, which represents the SAT problem, and *assignments*, which is a dictionary of variables that have already been assigned values. The function first identifies all unassigned variables and initializes their activity scores to 0. It then iterates over all clauses in the formula, and for each clause, it calculates the maximum decision level of the literals in the clause that have been assigned. For each literal in the clause, if the variable of the literal is unassigned, it increases the activity score of the variable based on the number of times it appeared in conflicts, divided by a decay factor raised to the power of the maximum decision level. The decay factor is set to 0.95 in this implementation. Finally, the function selects the variable with the highest activity score and randomly assigns it a value of True or False. This variable-value pair is then returned as the next branching decision. This approach ensures that the solver focuses on the most “interesting” parts of the problem, leading to more efficient search.



```
class Assignments(dict):
    """
    The assignments, also stores the current decision level.
    """
    def __init__(self):
        super().__init__()

        # the decision level
        self.dl = 0

        # Variable activities for VSIDS
        self.var_activities = {}

    def increment_activity(self, variable: int, increment: float):
        """
        Increment the activity of a variable.
        """
        if variable not in self.var_activities:
            self.var_activities[variable] = 0.0
        self.var_activities[variable] += increment

    # Code present under cdcl_solve method
    # Update activity scores of variables involved in the conflict
    max_dl = max(assignments[lit.variable].dl for lit in clause)
    for lit in clause:
        var = lit.variable
        if var in assignments:
            assignments.increment_activity(var, 1.0 / (math.pow(0.95, max_dl)))
```

Figure 4: Updating activity scores under *cdcl_solve*

In the above *Figure 4*, *max_dl* is the maximum decision level of the literals in the current conflicting clause. For each literal in the clause, if the variable of the literal is in the current assignments (i.e., it has been assigned a value), the activity score of the variable is incremented. The increment is inversely proportional to a decay factor (0.95 in this case) raised to the power of *max_dl*. This means that the activity score of a variable increase more if it is involved in a conflict at a higher decision level. The decay factor ensures that the activity scores of variables decrease over time unless they are involved in new conflicts. This mechanism helps the solver to adapt to the changing “importance” of variables as the search progresses.

C) Handling single empty clause scenario in CDCL

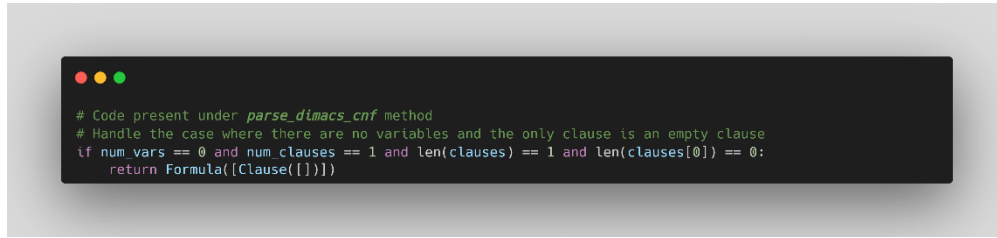


Figure 5: Handling single empty clause

Figure 5 handles the scenario where the formula has no variables, and the only clause is an empty clause. In the DIMACS CNF format, *num_vars* represents the total number of variables and *num_clauses* represent the total number of clauses in the formula. If *num_vars* is 0 and *num_clauses* are 1, it means that the formula has no variables and only one clause. If this single clause is empty (i.e., $\text{len}(\text{clauses}) == 1$ and $\text{len}(\text{clauses}[0]) == 0$), then the function returns a Formula with a single empty Clause. This scenario typically represents a trivially unsatisfiable formula, as an empty clause (a clause with no literals) is always false. This special case handling is important for the efficiency and correctness of the SAT solver.

Performance Analysis of CDCL SAT Solver

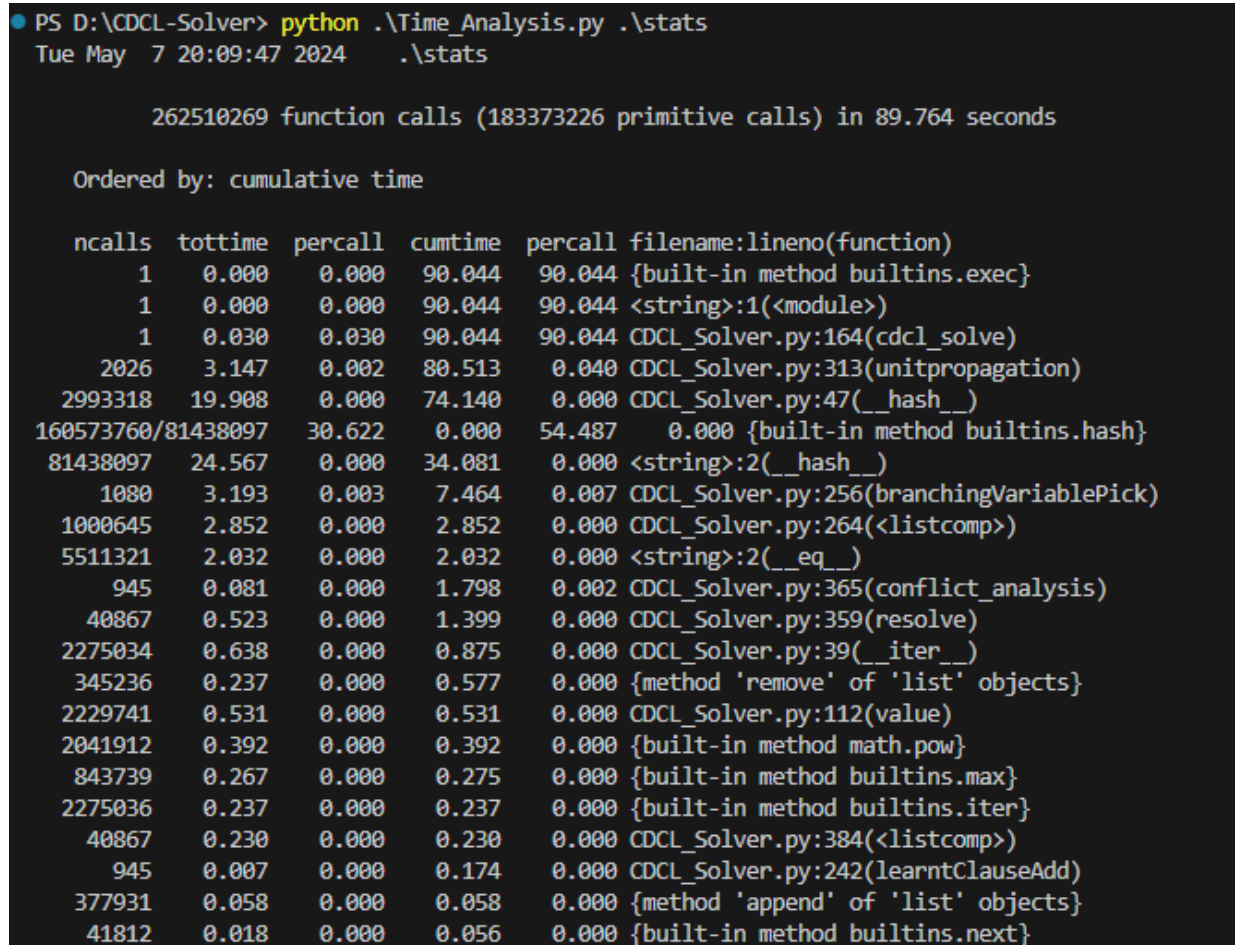


Figure 6: Performance analysis of a SAT testcase (cnfgen-php-10-10.cnf)

Based on *Figure 6*, we can make the following observations and analysis:

1. Bottleneck Functions:

- The *unitpropagation* function still takes a significant portion of the total time (80.513 cumulative time out of 90.044 seconds), suggesting that this function remains a potential bottleneck.
- The `__hash__` method of the Clause class continues to consume a substantial amount of time (74.140 cumulative time), indicating that the hashing operation for clauses is still a bottleneck.

2. Data Structures:

- The hash function from the built-in `builtins` module is called frequently, accounting for 54.487 cumulative time, which is likely due to the hashing operations in the Clause class and other data structures.
- The `remove` method of list objects is called 345,236 times, consuming 0.577 cumulative time, suggesting that removing elements from lists is still a recurring operation in the code.

3. Variable Selection and Branching:

- The *branchingVariablePick* function, which implements the VSIDS heuristic for variable selection, is called 1,080 times and takes 7.464 cumulative time, slightly higher than the previous profile.
- The *conflict_analysis* function, which analyzes conflicts and learns new clauses, is called 945 times and takes 1.798 cumulative time.

4. Clause Operations:

- The *resolve* function, which resolves two clauses on a variable, is called 40,867 times and takes 1.399 cumulative time.
- The *learntClauseAdd* function, which adds learned clauses to the formula, is called 945 times and takes 0.174 cumulative time.

Potential areas for optimization:

- Optimizing the *unitpropagation* function and the watched literal data structures.
- Improving the hashing operations for clauses or exploring alternative data structures like NumPy arrays.
- Optimizing the variable selection heuristic *branchingVariablePick* and the conflict analysis *conflict_analysis* functions.
- Reviewing the use of lists and the frequent removal operations *remove* method.
- Optimizing the clause resolution *resolve* and learned clause addition *learntClauseAdd* operations.

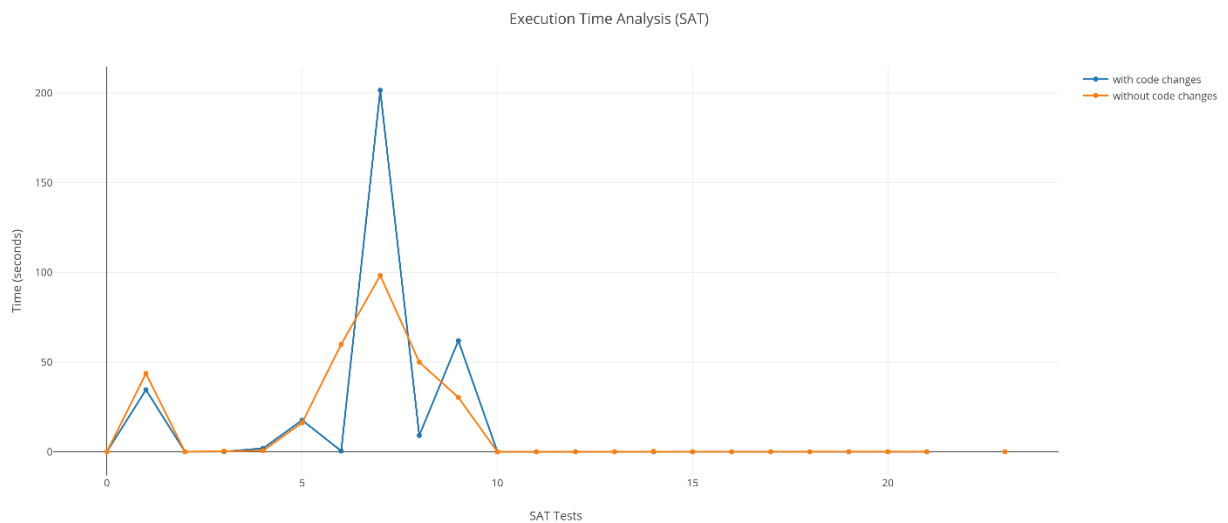
Note: The `StopIteration` error seen for the testcase `uf20-0106` can be resolved setting both `first_arithmetic_term` and `common_difference` to 1. By exploring the above potential areas for optimization, this problem could be solved.

Execution Time Analysis for SAT and UNSAT testcases

SAT Testcases

File Name	Number of Restarts	Execution Time with our code changes (seconds)	Execution Time without our code changes (seconds)
block0	0	0.00	0.00
cnfgen-php-10-10	1	34.56	43.65
elimredundant	0	0.01	0.0
prime121	0	0.12	0.41
prime169	0	1.93	0.63
prime841	0	17.62	16.18
prime961	0	0.45	59.89
prime1369	0	201.50	98.27
prime1681	0	9.10	50.01
prime1849	0	61.93	30.32
sat10	0	0.01	0.00
sat12	0	0.00	0.00
sqr10201	0	0.02	0.04
sqr10609	0	0.04	0.02
sqr11449	0	0.02	0.32
sqr1042441	0	0.03	0.04
uf20-0100	0	0.00	0.03
uf20-0101	0	0.00	0.02
uf20-0102	0	0.01	0.02
uf20-0103	0	0.01	0.03
uf20-0104	0	0.01	0.02
uf20-0105	0	0.01	0.07
uf20-0106	-	error	error
uf20-01000	0	0.01	0.03

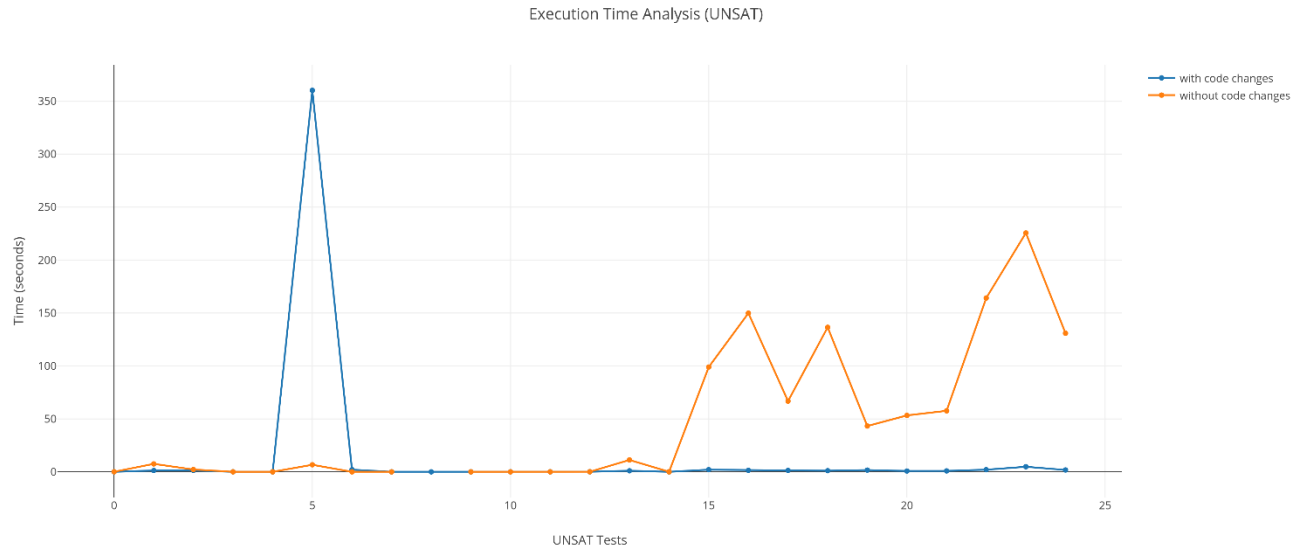
Table 1: Execution time and restart count for SAT testcases



UNSAT Testcases

File Name	Number of Restarts	Execution Time with our code changes (seconds)	Execution Time without our code changes (seconds)
add4	0	0.02	0.06
add8	0	1.24	7.48
cnfgen-parity-9	0	1.32	2.19
cnfgen-peb-pyramid-20	0	0.01	0.02
cnfgen-php-5-4	0	0.02	0.03
cnfgen-ram-4-3-10	4	360.28	6.67
cnfgen-tseitin-10-4	0	2.04	0.01
elimclash	0	0.00	0.00
false	0	0.00	error
full1	0	0.00	0.01
full3	0	0.01	0.01
full5	0	0.01	0.02
full7	0	0.06	0.07
ph6	0	0.99	11.20
unit7	0	0.01	0.04
uuf100-010	0	2.03	98.99
uuf100-012	0	1.39	149.83
uuf100-0117	0	1.32	66.78
uuf100-0120	0	1.08	136.46
uuf100-0130	0	1.64	43.26
uuf100-0147	0	0.78	53.26
uuf100-0151	0	0.89	57.60
uuf100-0161	0	2.01	164.16
uuf100-0175	1	4.80	225.59
uuf100-0182	0	1.76	130.86

Table 2: Execution time and restart count for UNSAT testcases



Conclusion

The CDCL SAT solver implementation has bottlenecks in the *unitpropagation* function and hashing operations for clauses. Potential optimizations include improving data structures, optimizing variable selection heuristics, conflict analysis, and clause operations. Reviewing the use of lists and frequent removal operations could also yield performance gains. Further profiling with diverse input instances may reveal additional optimization opportunities.

References

- [1] [CDCL-SAT-Solver](#)