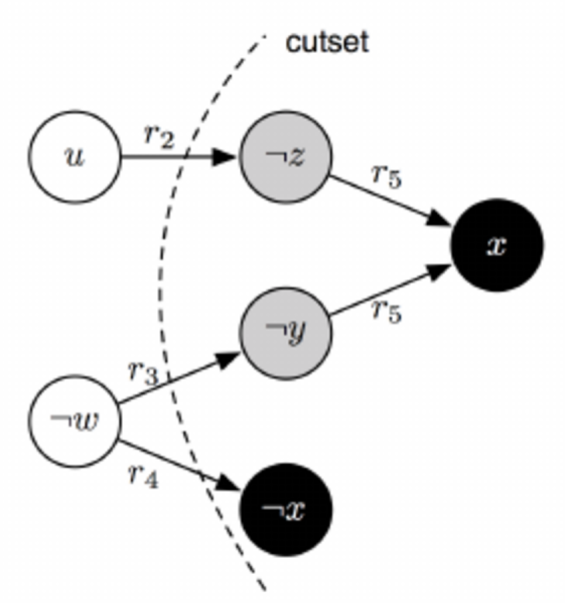
COL876: Special Topics in Formal Methods

Project 1: CDCL Based SAT Solver

October 2024

Abhinav Rajesh Shripad (2022CS11596)



Contents

1 Introduction

This report presents the design, implementation, and analysis of a SAT solver based on the Conflict-Driven Clause Learning (CDCL) procedure. The primary objective of this project is to develop an efficient SAT solver capable of checking the satisfiability of CNF formulas in DIMACS format. Our implementation aims to handle complex formulas with at least 150 variables.
Our project focuses on two critical components of the CDCL procedure:

- **Pick Branching Variable**: We discuss two heuristics for selecting the branching variable:
  - **Jeroslow Wang One Sided Solver** [1]: This method chooses the variable based on the Jeroslow-Wang rule, which assigns weights to variables depending on their occurrence in clauses, aiming to maximize the likelihood of satisfiability.
  - **Dynamic Largest Individual Sum Solver** [4]: This heuristic selects the variable based on the dynamic sum of individual clauses, favoring the variable with the largest influence.

  We compare the performance of these two heuristics and implement a hybrid heuristic that combines the strengths of both approaches.
- **Conflict Analysis**: For conflict analysis, we have implemented:
  - **Last UIP Cut**: This approach analyzes conflicts by tracing the conflict graph and selecting the last UIP for clause learning.

2 Notation

The following table summarizes the notations used in this report:

Symbol	Description
`TRUE`	Any variable/clause/formula which evaluates to True under the partial assignment
`FALSE`	Any variable/clause/formula which evaluates to False under the partial assignment
`UNASSIGN`	Any variable/clause/formula which evaluates to neither `TRUE` nor `FALSE`
n	Number of variables in a F_{CNF}
m	Number of clauses in a F_{CNF}
F	The Boolean formula in CNF format
r_c	Critical value of r at which phase transition occurs

Table 1: Table of Notations

3 Pick Branching Variable Heuristic

3.1 Dynamic Largest Individual Sum Solver

We implement the simplest version of **DLIS**, where for each `UNASSIGN` variable, we count the number of its occurrences with either parity in `UNASSIGN` clauses, and choose that variable and its assignment which occurs with maximum number.
We can think of it as, we append to our partial assignment a variable which decreases the number of `UNASSIGN` clauses the most. This also serves as an intuition for the heuristics.

3.2 Jeroslow Wang One Sided Solver

This is my favorite heuristic. Let S be the set of disjunctive clauses in F_{CNF}, we define the weight of S to be as

$$W(S) = \sum_p \frac{n_p}{2^p}$$

where n_p is the number of clauses in S with length p in S, in our Project p is always 3. We now define for each `UNASSIGN` literal x and its assignment, we define i $W(S_x^i)$ as weight of subset of S in which remains `UNASSIGN` after assigning x as i. We choose a literal x and its assignment i for branching such that it makes $W(S_x^i)$ maximum. Ties are resolved arbitrarily.
We can see that a clause with size p rules out exactly 2^{n-p} assignments from the possible set of truth assignments. Thus S rules out **at most** $\sum_p n_p 2^{n-p} = 2^n W(S)$. Thus we choose a literal and an assignment which minimizes the weight, thus ruling out a very less number of assignments, thus making it very likely to be **SAT**. [1]

4 Conflict Analysis Heuristic

4.1 Last UIP Cut

The Last UIP Cut is a crucial technique in conflict analysis for SAT solvers. When a conflict occurs during the solving process, the Last UIP Cut helps in determining which variables to backtrack to in order to learn a new clause that can prevent the same conflict from occurring again. The Last UIP is identified as the most recent point in the graph where a unique implication leads to the conflict. By backtracking to this point, the solver can derive a new clause that excludes the conflicting assignment, thus improving the efficiency of the search.

5 Implementation Details

The major points to discuss in the implementation is how to update the implication graph, unit propagation and conflict analysis. Inspiration for the implementation is taken from [2]

5.1 Updating the Implication Graph

The implication graph is updated in the `update_implication_graph` method:

```
def update_implication_graph(self, var, clause=None):
    node = self.implication_graph[var]
    node.value = self.assignments[var]
    node.level = self.decision_level
    if clause:
        for v in [abs(lit) for lit in clause if abs(lit) != var]:
            node.parents.append(self.implication_graph[v])
            self.implication_graph[v].children.append(node)
        node.clause = clause
```

This method updates the node corresponding to the assigned variable, setting its value and decision level. If the assignment results from unit propagation, it also updates parent-child relationships and stores the clause responsible for the implication.

5.2 Unit Propagation

Unit propagation is implemented in the `unit_propagation` method:

```
def unit_propagation(self):
    while True:
        propagation_queue = deque()
        for clause in self.clauses.union(self.learned_clauses):
            clause_value = self.evaluate_clause(clause)
            if clause_value == TRUE:
                continue
            if clause_value == FALSE:
                return clause
            else:
                is_unit, unit_literal = self.is_unit_clause(clause)
                if not is_unit:
                    continue
                propagation_pair = (unit_literal, clause)
                if propagation_pair not in propagation_queue:
                    propagation_queue.append(propagation_pair)
        if not propagation_queue:
            return None
```

```
        for prop_literal, clause in propagation_queue:
            prop_var = abs(prop_literal)
            self.assignments[prop_var] = TRUE if prop_literal > 0 else FALSE
            self.update_implication_graph(prop_var, clause=clause)
            try:
                self.propagation_history[self.decision_level].append(prop_literal)
            except KeyError:
                pass  # propagated at level 0
```

This method identifies `UNASSIGN` unit clauses, propagates their implications, and updates assignments and the implication graph. If a conflict is detected, it returns the conflicting clause.

5.3 Conflict Analysis

Conflict analysis is performed in the `analyze_conflict` method:

```
def analyze_conflict(self, conflict_clause):
        def latest_assigned_var(clause):
            for var in reversed(assign_history):
                if var in clause or -var in clause:
                    return var, [x for x in clause if abs(x) != abs(var)]

        if self.decision_level == 0:
            return -1, None

        assign_history = [self.decision_history[self.decision_level]] +
                    list(self.propagation_history[self.decision_level])

        pool_literals = conflict_clause
        processed_literals = set()
        current_level_literals = set()
        previous_level_literals = set()
        while True:
            for lit in pool_literals:
                if self.implication_graph[abs(lit)].level == self.decision_level:
                    current_level_literals.add(lit)
                else:
                    previous_level_literals.add(lit)

            if len(current_level_literals) == 1:
                break

            last_assigned, others = latest_assigned_var(current_level_literals)
            processed_literals.add(abs(last_assigned))
```

```
            current_level_literals = set(others)
            pool_clause = self.implication_graph[abs(last_assigned)].clause
            pool_literals = [l for l in pool_clause if abs(l) not in
            processed_literals] if pool_clause else []

        learned_clause =
            frozenset(current_level_literals.union(previous_level_literals))
        backtrack_level = max([self.implication_graph[abs(x)].level for
            x in previous_level_literals]) if previous_level_literals else
        self.decision_level - 1

        return backtrack_level, learned_clause
```

This method implements conflict-driven clause learning (CDCL). It analyzes the conflict by traversing the implication graph backwards from the conflict clause, identifies the Unique Implication Point (UIP), generates a learned clause, and determines the backtrack level.

6 Experiments

All experiments were done on the departmental BaadalVM Centos system to maintain uniformity for measuring the time taken by the solver. We applied a time-out of a maximum of 3600 seconds, i.e., 1 hour, on the solver for one particular problem. The solver was tested on 10 randomly generated SAT formulas for each value of r, leading to a total of 300 formulas.

6.1 Time Taken

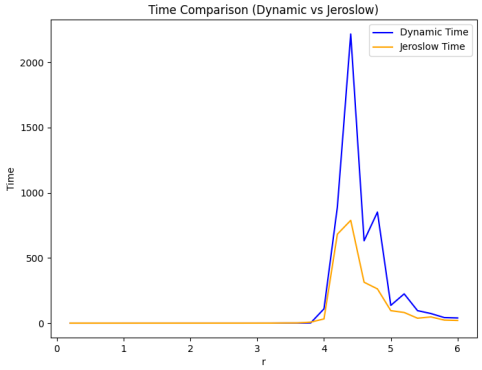


Figure 1: Average time taken per SAT formula vs r

6.2 Decision Count

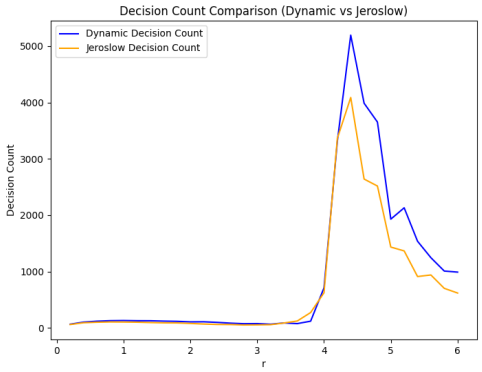


Figure 2: Average decision count per SAT formula vs r

6.3 Decisions per Unit Time

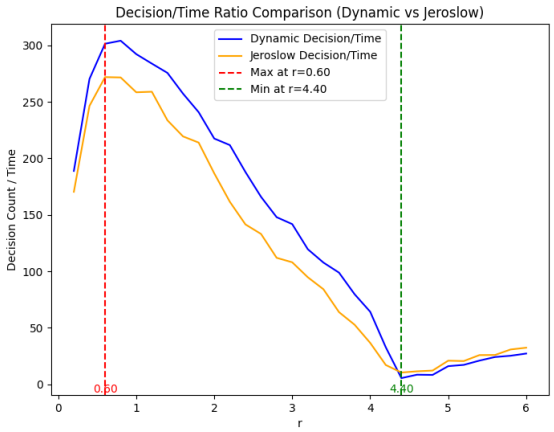


Figure 3: Average decision count per unit time vs r

6.4 Percentage of Satisfiable Assignments

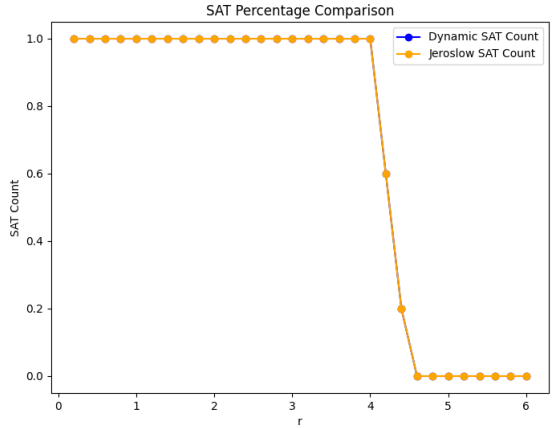


Figure 4: Percentage of satisfiable assignments vs r

7 Analysis

7.1 Phase Transition

We will provide an explanation of why a phase transition exists on the Satisfiability of the formulas and then provide a theoretical way to estimate the boundary of this critical behavior.

7.1.1 Why r_c exists ?

Let S be the set of all the satisfying assignments of F_{CNF} and similarly S' is for $F_{CNF} \cup c$ where c is a clause, then we can see that $S' \subset S$, thus increasing the length of the formula by keeping the total variables fixed, will mostly decrease the set of satisfying assignments. Thus we will reach a point where the set S is empty and F_{CNF} becomes `UNSAT`

7.1.2 Calculating r_c

Consider a fixed F_{CNF}, for each clause it reduces the solution space by a factor of $1-\frac{1}{2^k}$ where k is the number of variables in a single clause. If we assume, that each clause divides the space in same ratio, without any interaction with any other clause, then we can say the final solution space is of the size $2^n * (1-\frac{1}{2^k})^m$, if this is greater than 1 or not, we can find the critical value of r as

$$r_c = \frac{1}{k - \log_2(2^k - 1)}$$

This prediction is very close to the experimentally observed values of r_c for larger k. The following is the table from [3]

k	$\alpha_{c(theoretical)}$	$\alpha_{c(observed)}$
2	2.41	1.0
3	5.19	4.17 ± 0.05
4	10.74	9.75 ± 0.05
5	21.83	20.9 ± 0.1
6	44.01	43.2 ± 0.2

Table 2: Values of $\alpha_{c(theoretical)}$ and $\alpha_{c(observed)}$ for different k in k-SAT problems

7.2 Decision Count and Time Taken

We can see that as the r value approaches r_c from either side, there is a huge spike in the number of decision count and time taken by the solver. This can be explained by the fact that F_{CNF} in these regions are almost equally likely to be `SAT` or `UNSAT`, unlike lower r values, where due to less number of clauses, it is highly likely that we reach a satisfying assignment, and for larger r values, due

to huge number of clauses to satisfy, it is more likely we encounter a conflict quickly, thus the time taken and decision count is pretty less here. Author is unable to provide a mathematical derivation to approximate the time taken on average.

7.3 Decision Count per unit Time

Based on the graph, we can divide the behavior of the decision count per unit time (Decision/Time) into three regions:

- **For** $r \leq 1$:
  - In this region, both the Dynamic and Jerowslow heuristics show a rising trend in the Decision/Time ratio. The reason for this is the number of clauses in which a variable occurs on average is very less ($\frac{m}{n} \leq 1$), thus whenever there is a decision made, there are very less clauses which get affected by it, and very less propagation occurs, which means we need to make another decision very soon. One can see from the data, that number of decision count for $r \leq 1$ is approx 130, which means out of the 150 variables, we are picking 130 of these, and very less propagation is happening. Thus explaining the curve.
- **For** $1 < r < r_c$ **(Intermediate Region)**:
  - Here since there is an increase in number of clauses per variable r, we can conclude that a significant amount of time is spent in variable propagation and conflict resolution. With increase in number of clauses, each clause creates more opportunity for propagation (and not conflict, because in this region, `SAT` is more likely). So with increase in r, there is almost a linear decrease in decision count per unit time, as less number of decisions are needed.
- **For** $r > r_c$:
  - Here since it is more likely that the F_{CNF} is `UNSAT`, the clauses encountered after nr_c number of clauses are useless since, we have already encountered a conflict. So in this region we spent similar amount of time in propagation and conflict resolution. The slight increase in time with r can be attributed to the fact that more clauses are to be traversed.

This trend is not specific to any heuristic. One can observe this trend in the glucose 3 solver in pysat also.

8 Submission Details

The following files are included in this submission:

- `README.md`: A detailed overview of the project, including instructions for running the code, dependencies, and explanations of key components.
- `main.py`: The main execution file which integrates all components and drives the core functionality of the project.
- `solver.py`: Contains the implementation of the core algorithms or logic for solving the problem at hand.
- `branch_heuristics.py`: Includes specific heuristics used for branching decisions within the solver.
- `testcases`: A directory containing various test cases used to verify the correctness of the implementation.
- `benchmark.csv`: Data generated during benchmarking that reflects the performance of the solver on various test cases.
- `makefile`: Automates the process of compiling and running the project, ensuring all dependencies are handled.
- `requirements.txt`: Lists the external Python packages required to run the project.
- `Report.pdf`: This file itself
- `references.bib`: All the references/citations referred.

References

[1] Robert G Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1):167–187, 1990.

[2] Nitin Kedia. satispy, 2024.

[3] Scott Kirkpatrick and Bart Selman. Critical behavior in the satisfiability of random boolean expressions. *Science*, 264(5163):1297–1301, 1994.

[4] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.