ABCDCL - an implementation of CDCL with two watched literals in Rust

Course Project of DCC831 Theory and Practice of SMT Solving

Atila Carvalho & Bernardo Borges

January 27, 2025

Contents

1 Introduction

This report presents our course project for *DCC831 - Theory and Practice of SMT Solving*, where we implemented and evaluated a modern SAT solver based on the Conflict-Driven Clause Learning (CDCL) algorithm [1]. Through this implementation, we explored both the theoretical foundations and practical challenges of building an efficient SAT solver, gaining valuable insights into the complexities of state-of-the-art approaches.

The CDCL algorithm incorporates several key concepts covered in the course, such as unit propagation, clause learning, and non-chronological backtracking. Understanding these principles was essential not only for implementing the solver but also for debugging and refining its performance. Throughout the development process, we wrote extensive tests to verify correctness and experimented with different heuristics to optimize decision-making.

We chose Rust [2] as our implementation language due to its strong emphasis on memory safety, performance, and modern software engineering practices. Additionally, Rust is the primary language used in *Carcara* [3], a proof checker we have been working with, making it a natural choice for this project.

To evaluate our solver, we developed scripts that automatically download SAT benchmarks [4] in the DIMACS format and run them against both our implementation and the widely used *MiniSat* solver [6]. The results are logged and compared, providing an empirical analysis of our solver's performance and correctness.

2 Solver Design and Implementation

Our CDCL solver is designed with a modular architecture, focusing on clarity, correctness, and efficiency. The solver implements the Conflict-Driven Clause Learning (CDCL) algorithm, which consists of several key components: the search process, conflict analysis, backtracking, and clause learning. These components are supported by carefully chosen data structures and heuristics to ensure optimal performance.

2.1 CDCL Algorithm

The CDCL algorithm operates by iteratively assigning values to variables, propagating implications, and resolving conflicts. When a conflict arises, the solver performs conflict analysis to derive a learned clause, which is added to the clause database to prevent future conflicts. Backtracking is then used to undo decisions up to the appropriate decision level, and the search continues. This process repeats until either a satisfying assignment is found or the problem is proven unsatisfiable.

2.2 Data Structures

The solver relies on several key data structures:

- **Assignment**: Tracks the current assignment of variables, including their decision levels and antecedent (if any).
- **Clause**: Represents clauses in the formula, including learned clauses derived during conflict analysis.
- **Literal**: Encapsulates a variable and its polarity, used throughout the solver.

2.3 Code Structure

The core solver logic resides in `cdcl/mod.rs`, where the `Cdcl` struct implements is methods:

```rust
pub struct Cdcl<H: DecideHeuristic> {
    pub formula: Vec<Clause>,
    pub decision_level: usize,
    model: Vec<Option<Assignment>>,
    clauses_with_lit_watched: HashMap<Literal, HashSet<ClauseIndex>>,
    decide_heuristic: H,
}

impl<H: DecideHeuristic> Cdcl<H> {
    /// Main solving loop implementing the CDCL algorithm.
    /// Runs unit propagation, decision-making, conflict analysis, and backtracking.
    pub fn solve(&mut self) -> CdclResult { .. }

    /// Performs unit propagation to simplify clauses.
    /// Detects conflicts if a clause becomes empty.
    fn unit_propagation(&mut self, to_propagate: &mut Queue) ->
    ↪  UnitPropagationResult { .. }

    /// Analyzes conflicts to determine the backtrack level.
    /// Returns the decision level and learned clause if applicable.
    fn conflict_analysis(&self, conflict_clause_index: ClauseIndex) ->
    ↪  Option<(usize, Clause)> { .. }

    /// Backtracks to a given decision level, undoing assignments above it.
    fn backtrack(&mut self, b: usize) { .. }

    /// Chooses the next variable to assign using the heuristic.
    /// Panics if no unassigned variables remain.
    fn decide(&mut self) -> Literal { .. }
}
```

The solver requires an implementation of the `DecideHeuristic` trait

```rust
pub trait DecideHeuristic {
    /// Gets a random boolean
    fn next_polarity(&self) -> bool;
    /// Gets a random variable, if any exist
    fn next_variable(&self, model: &[Option<Assignment>]) -> Option<usize>;
}
```

This defines methods for variable selection, for which the `RandomDecideHeuristic` is used by default. Unit tests leverage both `RandomDecideHeuristic` and `MockDecideHeuristic` to validate correctness.

For parsing, we utilize the `dimacs` crate [9], integrated into `parser.rs`.

3 Testing and Evaluation

For the evaluation of our CDCL solver, we used a set of benchmark problems in DIMACS format from the SATLIB repository. Our solver was compared against MiniSat, a well-established SAT solver, in terms of correctness and runtime.

3.1 Experimental Setup

All experiments were conducted on a laptop with the following hardware and software configuration:

- **Operating System:** Arch Linux (Kernel 6.12.10)
- **Host:** ASUS VivoBook X512FJ
- **Processor:** Intel Core i7-8565U (8 threads, up to 4.60GHz)
- **RAM:** 8GB (usable: 7.8GB)
- **GPU:** Intel UHD Graphics 620 (Whiskey Lake) + NVIDIA GeForce MX230

3.2 Results

For each benchmark instance, we recorded the solver's result (SAT or UNSAT) and its execution time in milliseconds. The results were then compared against MiniSat to verify correctness and measure performance. The summarized results are presented in Table 1, comparing Avg Time to solve in *ms* for each benchmark and it's correctness overall.

Benchmark	ABCDCL Time	MiniSat Time	Results
dubois	5020	2	10 / 0 / 3 / 13
aim	109	1	72 / 0 / 0 / 72
jnh	275	2	50 / 0 / 0 / 50
pj1-tests/sat	1697	1	22 / 2 / 0 / 24
pj1-tests/unsat	4198	2	23 / 1 / 1 / 25

Table 1: Performance comparison of ABCDCL and MiniSat across benchmark families. Results is a counter of `(PASS/FAIL/TIME/TOTAL)`. The timeout used was 60 seconds.

3.3 Discussion

The majority of results obtained from our solver matched those of MiniSat, verifying its correctness, albeit for 3 cases that were parse errors, discussed in section 4. In terms of performance, our solver performed poorly against minisat, timing out in 3 cases of 'dubois' and 1 of 'pj1-tests/unsat' A potential reason for this performance difference is the naive (random) heuristic used in ours but also could be on memory management strategies. Future optimizations may focus on improving clause learning efficiency and branching heuristics to further reduce solving time while maintaining correctness.

4 Challenges and Lessons Learned

During the development and evaluation of our CDCL solver, we encountered several challenges that required careful debugging, algorithmic refinements, and adaptations to external dependencies. This section outlines the key difficulties faced and the lessons learned throughout the process.

One of the initial challenges was correctly following the CDCL algorithm as described in the reference materials [1]. The visual representations in the slides were sometimes

ambiguous, making it unclear what exact computations were being performed at certain stages. A notable example was the selection of the next watched literal, which we initially misunderstood due to implicit details, actually needs to *loop back around* sometimes, not explicitly illustrated in the reference diagrams.

Managing literals and their negations efficiently within the codebase also became a source of confusion. The tracking logic became increasingly complex, especially when handling clause propagation and backtracking. To address this, we refined our data structures and improved the organization of literals within the solver, with the separate **Literal** struct, making operations more transparent and reducing the likelihood of errors.

Debugging the solver proved to be another major challenge, particularly when implementing the decision-making functionality. The need to test with *determinism* a methods that uses randomness led us to introduce the `DecideHeuristic` abstraction. This allowed us to have both a standard version that made random assignments and a test-oriented version that could be precisely controlled. For the latter, we used the `mockall` crate, enabling us to define specific values for the `decide` function during unit testing, what allowed to write and diagnose issues related to decision strategies.

For more complex debugging scenarios, especially those involving intricate edge cases, we had to configure a Rust debugger using **LLDB** [8]. This setup allowed us to execute both the solver binary and individual unit tests in a step-by-step manner, analyzing control flow, and stepping into function calls, we were able to pinpoint the root causes of unexpected behavior like infinite loops and subtle logical errors that were difficult to catch through standard logging.

Additionally, we encountered a problem with the `dimacs` crate [9], which we used to parse input problems in the DIMACS format [10]. The issue arose when comments (lines starting with the character `c`) appeared in the middle of clause definitions, leading to parsing failures. Another edge case involved empty clauses, which also caused the parser to fail.

Ideally, we would have forked the crate and extended its functionality to correctly handle these cases. However, due to time constraints, we were unable to implement these fixes. As a result, certain test cases, namely `sat10.cnf`, `sat12.cnf` and `false.cnf`, are expected to fail due to parse errors.

Furthermore, we are currently working on implementing the VSIDS decision heuristic to improve solver performance. Unfortunately, this feature will not be ready in time for this report, but we plan to update the repository once it is completed.

5 Conclusion and Future Work

This project proved to be significantly more challenging than anticipated, primarily due to the nuanced balance between correctness and efficiency. Through the process, we gained a deep understanding of both the CDCL algorithm and the intricacies of implementing it in Rust. Despite the pressure, we enjoyed developing creative solutions, such as implementing unit tests, mocks, and scripts, comparing our solver with an existing one [6], and overcoming challenges with the parser crate. We were heavily inspired by a Python implementation [11], and in understanding its code, we forked and enhanced it by adding strict typing, which was later merged into the original repository `https://github.com/Kienyew/CDCL-SAT-Solver-from-Scratch/pull/1`.

For future work, we plan to further refine the solver by implementing the VSIDS

decision heuristic, which is expected to enhance performance. We also aim to fix the issues related to the DIMACS parser, ensuring that the solver can handle all edge cases correctly. Overall, this project was a rewarding challenge that deepened our understanding of both SAT solving and Rust programming.

References

[1] A. Oliveras, E. Rodriguez-Carbonell "From DPLL to CDCL SAT solvers", January of 2025, Available at `https://www.cs.upc.edu/~oliveras/LAI/cdcl.pdf`.

[2] "The Rust Reference", January of 2025, Available at `https://doc.rust-lang.org/reference/index.html`.

[3] B. Andreotti, H. Lachnitt, H. Barbosa "Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format", April of 2023, Available at `https://link.springer.com/chapter/10.1007/978-3-031-30823-9_19`.

[4] "SATLIB - Benchmark Problems", January of 2025, Available at `https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`.

[5] B. Borges, A. Carvalho "Atila - Bernardo - CDCL SAT Solver", January of 2025, Available at `https://github.com/bernborgess/abcdcl`.

[6] N. Eén, N. Sörensson "The MiniSat Page", January of 2025, Available at `http://minisat.se/MiniSat.html`.

[7] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik "Chaff: Engineering an Efficient SAT Solver", January of 2025, Available at `https://www.princeton.edu/~chaff/publication/DAC2001v56.pdf`.

[8] "The LLDB Debugger", January of 2025, Available at `https://lldb.llvm.org`.

[9] "Crate dimacs - The parser facility for parsing .cnf and .sat files as specified in the DIMACS format specification", January of 2025, Available at `https://docs.rs/dimacs/0.2.0/dimacs/index.html`.

[10] "DIMACS CNF Format", January of 2025, Available at `https://satcompetition.github.io//2009/format-benchmarks2009.html`.

[11] "CDCL SAT Solver from Scratch in Python", January of 2025, Available at `https://github.com/Kienyew/CDCL-SAT-Solver-from-Scratch`.