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

This section should describe the architecture and design choices of your CDCL solver. Some topics to cover:

- Description of the CDCL algorithm: search process, conflict analysis, backtracking, clause learning.
- Key data structures used in the solver (e.g., decision stack, implication graph, learned clauses).
- Heuristics implemented (e.g., variable selection, decision ordering).
- Optimizations applied (e.g., restarts, watched literals, UIP learning).

3 Experimental Setup

In this section, describe the experimental setup used to evaluate the solver. Include:

- Description of the SAT instances used in the experiments (e.g., DIMACS benchmark formats, problem types).
- Hardware and software environment for running experiments (e.g., processor, memory, operating system, libraries).
- Metrics used for evaluating solver performance (e.g., runtime, number of conflicts, memory usage, number of clauses learned).

4 Results and Evaluation

This section should present the results of your experiments and analyze the performance of your solver. Topics to include:

- Summary of experimental results with tables and figures (e.g., performance on different benchmarks, comparison with other solvers if applicable).
- Analysis of the solver's strengths and weaknesses based on the results.
- Discussion on the impact of various optimizations and heuristics on solver performance.

5 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. To resolve this, we had to fork the crate and extend its functionality to correctly handle comments interspersed within clauses. Another edge case is where an empty clause was also causing the parser to fail. Our modifications ensured that such cases were handled correctly, preventing unintended crashes.

6 Conclusion and Future Work

In the conclusion, summarize the key findings and suggest potential improvements:

- Recap of the objectives and how they were achieved.
- Summary of the solver's performance and main takeaways.
- Suggestions for future work (e.g., improvements to heuristics, additional optimizations, new features to implement).

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`.

A Appendix: Source Code

Include the relevant source code snippets or provide a link to the full implementation if it's hosted in a public repository (e.g., GitHub). For code snippets:

```rust
fn main() {
    println!("Hello, CDCL Solver!");
    // Your implementation here
}
```