CSCI2951O Project 1: Vehicle Customization

Name: Richard Tang, cslogin: rtang26, screenname: ginakonda

March 9, 2024

1 Introduction

In this project, I implemented a CDCL-based SAT solver in Rust. The SAT solver attempts the following optimizations: 2-watched literals, 1st-UIP clause learning and non-chronological backtracking, EVSIDS, Glucose-based clause deletion, Conflict clause minimization, and Luby-based restarts.

2 Implementation

To save space, the actual algorithm is detailed in the appendix, as well as some discussion on programming language choice (see A.1 and A.2). Some general implementation notes:

- Variables are mapped from the file values to an internal solver number, then mapped back on solution emission; this is done to prevent issues about gaps in numbering (as in this Edstem post).
- Literals are assigned as such: `lit = 2 * var + sign`. This allows all literals to exist from $[0, n_{2\text{vars}+1})$, which provides easy indexing into vectors.
- The algorithm does not implement pure literal elimination. From literature review (and this corresponding StackOverflow post), none of the major SAT solvers (PicoSAT, MiniSat, Chaff, etc) implement PLE for performance reasons.

I attempted various optimizations to improve performance:

- **CDCL**: Instead of standard DPLL, I attempted to implement CDCL, with conflict analysis and clause learning. In particular, I maintained

an trail of assigned variables, and their assignment reasons (either decision or propagating clause); then, on conflicts, I inspected the trail to find the first UIP (as defined in [9] and [6]). Most papers I referenced, however, didn't actually explain how to find the actual UIP, instead abstractly describing implication graphs; it turns out it's just a backtrack through the trail (a la reverse-BFS) to find the "dominator node" at the conflict decision level. I referenced the algorithm presented in MiniSat [5].

  - This also included non-chronological backtracking, which just popped off the trail and un-assigned variables until a desired decision level.

- **2-Watched Literals**: Most references described how boolean constraint propagation took up the majority of runtime (which I eventually verified using `flamegraph`s; see B), so I attempted watched literals to optimize this. [6] provided a good initial introduction to the algorithm; I then tried to implement the PicoSAT version with their pointer optimizations [3], but this proved too difficult to understand (and Rust didn't provide many required bit-packing language features in C/C++), so I again referenced the MiniSat[5] implementation, which maintained an occurrence list of watching clauses for *each literal*; by ensuring the watched clauses were always indices 0 and 1, it optimized for access.

- **Glucose-based clause deletion**: After implementing CDCL, I noticed my learnt clause database kept growing, which made BCP incredibly slow (after a minute, `C1597_024` had 1597 constraints and over 200k learnt clauses). Thus, I searched for clause deletion policies, eventually settling on Glucose's [1] LBD policy (which recorded the number of unique decision levels in each learnt clause).

- **Conflict Clause Minimization**: Like clause deletion, I noticed that learnt clauses had a large size (averaging ≈ 25 per clause); to reduce this, I implemented conflict clause minimization from a follow-up MiniSat paper [8] using self-subsuming resolution (essentially, if learned literals were implied by other literals or decisions, they are redundant), shrinking learnt clauses to average ≈ 8 per clause.

- **EVSIDS**: For branch decision heuristics, I decided to implement EVSIDS. I originally wanted to implement VSIDS, but it involved a significant amount of iteration (for each update!); moreover, the actual EVSIDS

implementation turned out to be quite easier. For decision heuristics in general, I referenced Chaff [7], MiniSat [5], and this survey paper [4].

- **Luby-based restarts**: I finally implemented Luby restarts; I wanted to try more specified in this paper [2], as well as Glucose-based restarts [1], but unfortunately did not have enough time.

2.1 Performance

CDCL and 2-watch literals alone, for me, were insufficient to pass all clauses in the allotted time. Glucose-based clause deletion, combined with conflict clause minimization, increased performance by up to 350%, but was still not sufficient. EVSIDS (instead of random assignment) further improved performance by around 300%, finally allowing me to pass all clauses within the time limit. Tuning parameters on clause deletion, sometimes randomizing branch decisions and polarity, etc. did not provide a significant improvement.

I tried implementing Luby restarts to speed up cases in which search entered a poor branch. Because this caused a lot of restarts on UNSAT instances, though, it ended up slowing down my program, so I disabled it by default.

I tried using multiple solver instances with different parameters and randomized decisions to find instances faster, but I guess my algorithm is too deterministic (or too easily falls into the same local minima), as this provided no performance improvement.

Besides algorithmic optimizations, I tried to use Rust's profile-guided optimization (PGO) to optimize compilation based on production examples (I used C1065_064.cnf, U50_1065_038.cnf, and U50_4450_035.cnf); however, this caused a slowdown by around 50%, so I disabled it.

2.2 Bugs

Given its perhaps surprisingly slow performance, it's likely that I have latent implementation errors (which are *hopefully* just performance, not correctness, issues). Here are some (mostly correctness) bugs that I discovered when implementing:

- Due to my literal representation, I often got confused with the actual literal values (it didn't help that I had a custom boolean represen-

tation, where `True == 0`...); many times I forgot to flip the polarity when assigning.

- I forgot to un-assign values on backtracking, sometimes causing their reason clauses to be incorrectly filled on conflict analysis.
- In conflict analysis, I originally assigned *unseen* literals as asserting literals, causing un-related reasons to be visited.
- I often failed to maintain invariants within the watch lists; watched values were supposed to be index 0 and 1, but I inconsistently changed clauses, causing duplicated and overridden literals. I also forgot to delete watchers, slowing down propagation.
- In clause deletion, I forgot to calculate LBD, and then I sorted it in reverse, causing the most *important* clauses to be deleted.
- In EVSIDS computation, my floating point representation often overflowed, causing values to hit `inf` or `NaN`. This rendered EVSIDS values useless.
- So many off by ones! In decision level, 1-UIP backtracking, 2-watch literals maintenance, etc. Indexing sucks :(

3 Conclusion

I spent roughly 40 hours implementing, debugging, and optimizing this project. Getting a concussion mid-way greatly slowed down progress and made understanding certain algorithms—looking at you, 1-UIP conflict analysis—exceedingly difficult, but it was super fun to implement, and I'd like to keep optimizing in the future.

A Implementation notes

A.1 Main algorithm

The SAT solver consists of one large loop implementing the DPLL (CDCL) algorithm. Unit clauses are repeatedly propagated:

- If no conflicts detected and all variables are assigned, we return SAT; otherwise, we decide the next variable using some heuristic, assign that variable, and re-propagate.
- If a conflict is detected, analyze the cause for the conflict, backtrack to the first UIP's decision level, and add that learned clause to the clause database. If the clause database is too large, clean it according to some policy. Re-propagate

If too many conflicts occur, there is an option to restart the search.

The algorithm was implemented following mostly Serdar's lecture notes, the paper "On the Unreasonable Effectiveness of SAT Solvers" [6], the original Princeton Chaff paper [7], and the MiniSat paper [5].

A.2 Programming language choice

I implemented the SAT solver in Rust. I was originally going to implement in C++, but I chose Rust for a few reasons:

1. Rust provides a rich ecosystem of both standard library and external "crates", with all sorts of functionality from custom hash functions to loggers to arena allocators. While C++ has a similar ecosystem, integrating it with a build system portably and linking during compilation proved to be a huge headache.
2. In general, it's been too long since I've set up a CMake/C++ project; moreover, due to the inherent memory unsafety of C++, influency would result in many potentially terminal bugs.

In general, this was a positive choice for me; Rust's strict ownership system and borrow checker caught many of my programming mistakes or invalid memory accesses. As a result, on compilation, I didn't encounter any(!) segfaults or undefined behavior from invalid variable modification (this is not to say I didn't have bugs, only that they were all actual algorithm implementation bugs).

However, Rust's borrow checker did pose a nuisance in some cases. In particular, my `CDCLSolver` object had many unrelated fields to store information (e.g. literal occurrence lists, variable activity, etc.), to which simultaneous access should pose no problem; however, because Rust's borrow checker forbids holding multiple mutable references at once, the compiler did not allow this to compile. I experimented with `RefCell`s and interior mutability, but due to its complexity (and associated runtime cost), I chose not to use them. In the end, I ended up making copies of certain values then re-assigning them if I did need multiple mutable accesses.

Patterns like this might suggest that Rust is still less performant than C/C++ in extreme cases, where the borrow checker forbids correct optimizations.

- Regardless, I think the developer experience of Rust far exceeds anything offered by C/C++.

B Flamegraphs

I used flamegraphs to find bottlenecks within my code; as expected, much of the time was spent in BCP. I didn't have time to completely re-factor my code or try different optimizations, but in the future I'd like to attempt.

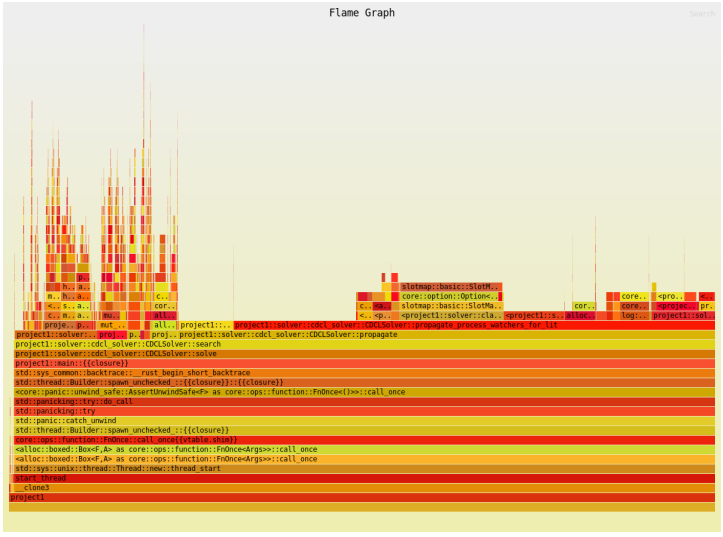

Figure 1: Flamegraph for U50_4450_035.cnf. Note how `CDCLSolver::propagate` occupies almost 80% of the execution time.

References

[1] Gilles Audemard and Laurent Simon. On the glucose sat solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, 2018.

[2] Armin Biere. Adaptive restart strategies for conflict driven sat solvers. In Hans Kleine Büning and Xishun Zhao, editors, *SAT*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008.

[3] Armin Biere. Picosat essentials. *J. Satisf. Boolean Model. Comput.*, 4(2-4):75–97, 2008.

[4] Armin Biere and Andreas Fröhlich. Evaluating cdcl variable scoring schemes. In Marijn Heule and Sean A. Weaver, editors, *SAT*, volume 9340 of *Lecture Notes in Computer Science*, pages 405–422. Springer, 2015.

[5] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[6] Vijay Ganesh and Moshe Y. Vardi. *On the Unreasonable Effectiveness of SAT Solvers*, page 547–566.

[7] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engeneering an efficient SAT solver. In *Design Automation Conf.*, 2001.

[8] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.

[9] Lintao Zhang, C.F. Madigan, M.H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, pages 279–285, 2001.