CSCI 2951-O: Mass Customization

Komron Aripov
cslogin: karipov
screenname: ghazal

March 1, 2024

Overview

This project was a massive process of trial and error that culminated in what I think is some of the most exciting code I've written in a while. I also spent a large chunk of the time on re-writing code, completely throwing out sections of it and making important decisions on where to focus my time.

Over the course of the month, I spent close to **60 hours** working on this project.

1 Rough Timeline

Week 0-1: I initially completed my project using **pure Python**. After attending lectures, I believed performance gains would primarily come from heuristics and algorithmic improvements. However, I found certain code sections needed greater efficiency than Python could provide.

Week 1-3: I then rewrote my functions in **Numpy**, expecting its C-based subroutines to speed up my solver. Despite Numpy's widespread use in major ML libraries, this approach failed mainly because array operations frequently reallocate data. I also explored **Numba** for JIT-compiling Python code, but its limited support and the extensive modifications required made it impractical.

Week 3-4: Ultimately, I switched to **Julia**, a language similar in syntax to Python but with built-in, fully supported JIT compilation. Despite the challenges of learning a new language on the fly, this change resulted in the fastest solver.

2 Important Optimizations

2.1 Data Structures

Recursion sucks :(Initially, I used recursive DPLL, which was inefficient due to the excessive stack frame allocations and data movement caused by recursion. To improve efficiency, I shifted to a FIFO **stack-based approach**, which tracks decisions and assignments without the overhead of recursion.

Why iterate over all the clauses? My early method involved naively modifying my clause set by removing variables and deleting clauses, which only worked for simple instances due to its slowness. The key was maintaining an immutable clause set and using **occurrence lists** to track literals and the clauses in which they exist. SAT solvers spend 90% of their time doing BCP, so this had a significant effect. I also spent a week trying to implement **2-watched literals**, but they ended up being very buggy.

Profiling! Profiling revealed Julia spent considerable time on **garbage collection**, exacerbated by functions like `map` and `sort` that frequently reallocate memory. I had to rewrite these using for-loops or used more efficient algorithms (e.g. quicksort). I switched to using `Int16` for my literal types instead of the default `Int64`, significantly reducing the garbage collection workload and improving overall efficiency.

2.2 Algorithms

Heuristics come later. Heuristics only work as long as your general BCP and DPLL algorithms are quick. I implemented **MOMS** with my initial python code and because BCP was abysmally slow, it did not help solve more instances at all.

Dynamic vs Static? I realized that in general, the trade-off to recomputing any heuristic function leaned heavily towards more static analyses. I implemented **Jeroslow-Wang** which would recompute at every decision level. This grinded my DPLL to a halt. I instead switched to an initial static analysis and re-computed once a large number of conflicts were encountered.

PLE – extra work for no reason. I played around with Pure Literal Elimination, but its inclusion / exclusion didn't result in any extra performance. In the end, I just removed it from my codebase altogether.

3 Takeaways

There was a lot of things I wish I did differently. Mainly, I spent a large amount of time focusing on rewriting code, rather than testing out optimizations. I did not prioritize efficiently, and gunned immediately for a fast solution – with two-watched literals, for example. After spending way too much time trying to make things work, I fell back on a simpler solution to at least have something working.

This was both the most challenging and rewarding in-class project I've done at Brown. I also think I'd love to write a full CDCL SAT solver. With my luck, I'll learn yet another language, maybe even Rust :)

References

[1] Matthew W. Moskewicz et al. Chaff: Engineering an Efficient SAT Solver. `https://www.princeton.edu/~chaff/publication/DAC2001v56.pdf`, 2001. Accessed: 2024-02-29.

[2] Daniel Geschwendery. Watched Literals and Restarts in MiniSAT. `https://cse.unl.edu/~choueiry/S17-235H/files/SATslides07.pdf`, 2017. Accessed: 2024-02-29.

[3] Christoph Kreitz. Cornell CS 4860 Applied Logic Lecture Notes. `https://www.cs.cornell.edu/courses/cs4860/2009sp/lec-04.pdf`, 2009. Accessed: 2024-02-29.

[4] Rohan Menezes. University of Pennsylvania CIS 1890 Lecture Notes. `https://www.cis.upenn.edu/~cis1890/files/Lecture3.pdf`, 2023. Accessed: 2024-02-29.