Mass Customization Report

Name: Julian Dai; CS Login: jkdai; Pseudonym: monkey

Stragey

My implementation sticks faithfully to the classic DPLL algorithm discussed in class, consisting of:

- Unit Propagation (UP)
- Pure Literal Elimination (PLE)
- Branching + heuristics (e.g. DLCS, DLIS)

Initial Implementation

The initial implementation consisted of a recursive DPLL algorithm implementing UP, PLE, and random branching in Python with simplistic data structures (i.e. only a list of clauses) that were copied on each branch. With this approach, I achieved a score of 4800, with 14/22 of the problems unsolved in the allotted 5 minutes. With some low-hanging engineering improvements (e.g. using list comprehension to copy data structures instead of copy()/deepcopy()), the score improved significantly — for example C168_128 decreased from 26.50s to 5.25s and C181_3151 decreased from 13.57s to 7.33s.

(Attempted) Optimizations

I will now discuss (most of) the optimizations I implemented to varying degrees of success.

A Search Heuristic & Intelligent Data Structures

The first major improvement in my SAT solver came from 2 things: 1) Implementing DLCS/DLIS as a search heuristic 2) Improving the efficiency of unit propagation and pure literal elimination via intelligent data structures: - unit_literals: list - lit2clause: dict[int, set(int)]

While these data structures can be maintained with relatively low cost and speed up UP, PLE, and DLCS/DLIS, they must also be copied on every branching. However, the experimental results showed that the benefits introduced by these data structured outweighed the cost. This implementation produced a score of 2377.65s, with 6 unsolved CNF instances.

Additional Optimizations

Unsure what exactly to do next, I profiled my code by tracking time spent in relevant functions (using an in-program method as opposed to a CPU polling method like flamegrpahs), observing that my SAT Solver unsurprisingly spent the largest amount of times performing unit propagation and copying data structures. To further optimize my code, I attempted an augmentation of my existing python implementation with watched literals, which theoretically speeds up UP and removes the need for the copying of some data structures.

Watched Literals

In order to support efficient backtracking with watched literals, I scaled back PLE to run only during pre-processing rather than on every loop of my solver. I'm unsure if I simply engineered things incorrectly (I passed some instances comparably fast compared to the implementation without watched literals), but overall, this implementation scored consistently lower on multiple instances.

CDCL

I also tried a python CDCL implementation, following the MiniSat C++ implementation. It was very interesting to learn about in more detail, especially the method for picking learned clauses (e.g. using and implementing FirstUIP). However, after achieving a functioning implementation including FirstUIP, VSIDS,

and watched literals (as verified by a subset of the input problems) but wihtout yet implementing random restarts or clause deletion, I realized, similar to the watched literals case, my implementation of CDCL was in general slower than my vanilla DPLL algorithm, which led me to think that there are likely bugs in my implementation slowing it down — and so I gave up on this and instead tried moving my implementation to Rust.

Python vs Rust

Moving to Rust unsurprisingly became the 2nd major improvement of my algorithm. With compiler optimizations turned on (i.e. compiling in release mode), the Rust implementation completed specific instances up to 10 times faster than the Python implementation, for a total score of ~1602. With the exception of one instance, the 6 instances that remained unsolved in the time constraints were also unable to be solved by the Rust implementation.

Some more optimizations I tried in Rust: - Using Vectors rather than HashSets and HashMaps: An optimization that took quite a bit of effort in restructuring my implementation, yet yielded worse scores than the hashing-based implementation. - Using FXHash: A drop-in replacement for HashMaps and HashSets in Rust that is a bit faster. Shaved off ~16 seconds across the 17 solved instances for a new score of ~1584.

Parallelizing

Parallelization (implemented hastily on 2/28) is the final improvement. My implementations assumes 8 cores, but the general idea applies to n cores. Using multiple cores to process things faster in 2 ways:

1) Parallelizing heuristics

I didn't have much success in profiling specific instances to determine what heuristic to use, so instead, I ran multiple threads that each ran a different hueristic (DLIS, DLCS, RandDLIS, RandDLCS, and Hybrid, which would randomly pick DLIS or DLCS with 50% probability). The program terminates as soon as one thread returns SAT/UNSAT, as that thread has searched the entire space.

2) Parallelizing search space

Another parallelization strategy is to split up the space that each thread searches. To do this effectively, my implementation retrieves the top 3 literals according to the DLCS metric and then distributes all possible assignments (i.e. 2^3 = 8) across all threads, where each thread gets a unique assignment. In this way, the entire space will be searched. DLCS is used because the positive and negative occurence of each literal must be used to ensure completeness, so we want to maximize the total occurences of the positive and negative literal as opposed to a single polarity.

Note: Since both methods relies on luck, the results are not entirely representative, but would be more so if an average were taken across multiple runs. However, preliminary runs show that both methods yield significant speedups.

Takeaways

- Start in a high-performance language: The speed-ups are worth the PITA for coding in Rust (at least compared to a language like Python)
- There's always multiple layers of optimization: e.g. language, paralellization, randomization
- Sticking with CDCL: I guess it's easy to say this in retrospect, but I wish I stuck to my CDCL implementation. It would have been very satisfying to solve any of the 5 instances by solver couldn't complete.

Time spent: ~ 30 hours