DPLL Solver Implementation

Christian Armstrong (camstr8, Barron)
Taj Gillin (tgillin, squirtle)

February 28, 2025

1 Overview

In this project, we have been contracted by none other than ECC to solve their SAT problems. We have been given various CNF files, and our goal is to develop a solver based on DPLL to find a solution or determine if they are infeasible efficiently.

2 Implementation details

Our original solution was coded in Python. However, Python is slow. We have since switched to C++, and implementing the same algorithm as Python gave a speedup of one to two orders of magnitude. Overall, we worked through six versions of our solver, each version improving on the last.

2.1 Version 1

The first version implemented was a basic DPLL solver. It represents the formula as a vector of vector of ints. Each int is a literal, and these literals are groups into clauses as vectors. These clauses (vectors of ints) are then grouped together with another vector, giving us our formula. Our assignment is a vector of variables. Each index corresponds to a variable, with each value a enum representing unassigned, positive, or negative assignments. It is initialized to unassigned and populated as the algorithm progresses. Using this representation, we do not modify the clause representation. When we assign a variable, we simply add it to the assignment vector. When branching/splitting on a variable, we copy the assignment, allowing for straight-forward backtracking by simply returning.

2.2 Version 2

For the second version, we implemented watched literals. This removes a big time sink during unit propagation, evaluating every clause. This was implemented as discussed in class, watching two unassigned literals and attempting to find a new literal when the watched value is assigned. Overall, this led to a speedup of around 5x.

2.3 Version 3

For the third version, we implemented a few changes. The first was formula preprocessing, where we removed any clauses with tautologies (as they will always be satisfied). Next, we implemented a variable-to-value alias before running the solver. The goal of this was to assign the variable numbers, which can be any number and not the lowest possible values, aliases for when we solve. They were assigned numbers 1 to n, where n is the number of unique variables, solved, then mapped back when done. This led to an enormous speedup, as our vectors were now of minimal size and able to load/cache extremely efficiently. The last change was implementing MOMS (maximum occurrences on minimum size clauses) as our branching heuristic. Overall, these changes resulted in a speedup of around 20x.

2.4 Version 4

For the fourth version, the only change we made was implementing polarity selection when branching. That is, instead of always choosing the true branch to test first, the solver selects whichever polarity (positive or negative) appears more. This helps us find solutions faster, and performs marginally better on some instances. For the most part, it performs relatively similarly.

2.5 Version 5

For the fifth version, we optimized the UNSAT detection. This was done by only checking clauses that were modified during assignment and doing so rapidly. This removes redundant evaluation and resulted in about 20% improvement in some instances.

2.6 Version 6

Lastly, we tried some compiler optimizations, which didn't help a ton, but shaved off a little bit of time.

3 Other attempts

There were a few other methods we attempted that hurt our implementation. First, we attempted to use the phase transition (ratio of variables to clauses) to determine when a clause is likely unsat and use no heuristic instead of MOMS. The thought process was that MOMS requires extra calculations, and this will slow down the UNSAT implementation, which must explore every branch. In practice, however, MOMS triggered more unit propagation and allowed for more inference steps, speeding up the implementation over using no heuristic.

Secondly, we tried to implement some randomness to our heuristic using something WalkSAT inspired MOMS heuristic. With some hyperparameter epsilon, it would randomly either pick the best variable calculated from MOMS, or it would pick a random variable from the shortest found clauses. We tried to combine this with a simple implementation of randomized restarts. It might have worked if we continued tuning the hyperparameters, using something like grid search to find the best combination, but it simply did not speed up the SAT problems and led to decreases in speed in the UNSAT.

Third, we tried CDCL, but we simply couldn't get it to work efficiently. I believe that with randomized restarts and an effective heuristic, it may perform well. However, our implemented heuristic, MOMS, did not have a random element, making randomized restarts ineffective.

Lastly, we tried other heuristics by themselves. Jeroslow-Wang in particular, proved very efficient on some problems but incredibly slow on most. No other heuristic was able to perform well on all the problems. A future direction would be developing a portfolio of solvers, possibly solving in parallel, all using different heuristics. With this portfolio, different classes of problems could be efficiently solved by a heuristic that performs well on it.

4 Summary

Overal, our solver is quite fast on most cases and we are proud of the result. There are a few cases that we solve that are slow compared to others, and we were able to solve some of these faster with different heuristics, so a good future direction would be building a portfolio of solvers to approach different problem classes. Additionally, there appear to be a few problems that no one is able to solve. Holistically looking at the problems, they are either very long, very complex, or both. We believe that successfully implementing CDCL, along with randomized restarts and a randomized heuristic, allowing for learning across restarts, may be able to help tackle these problems.

5 Extra Figures

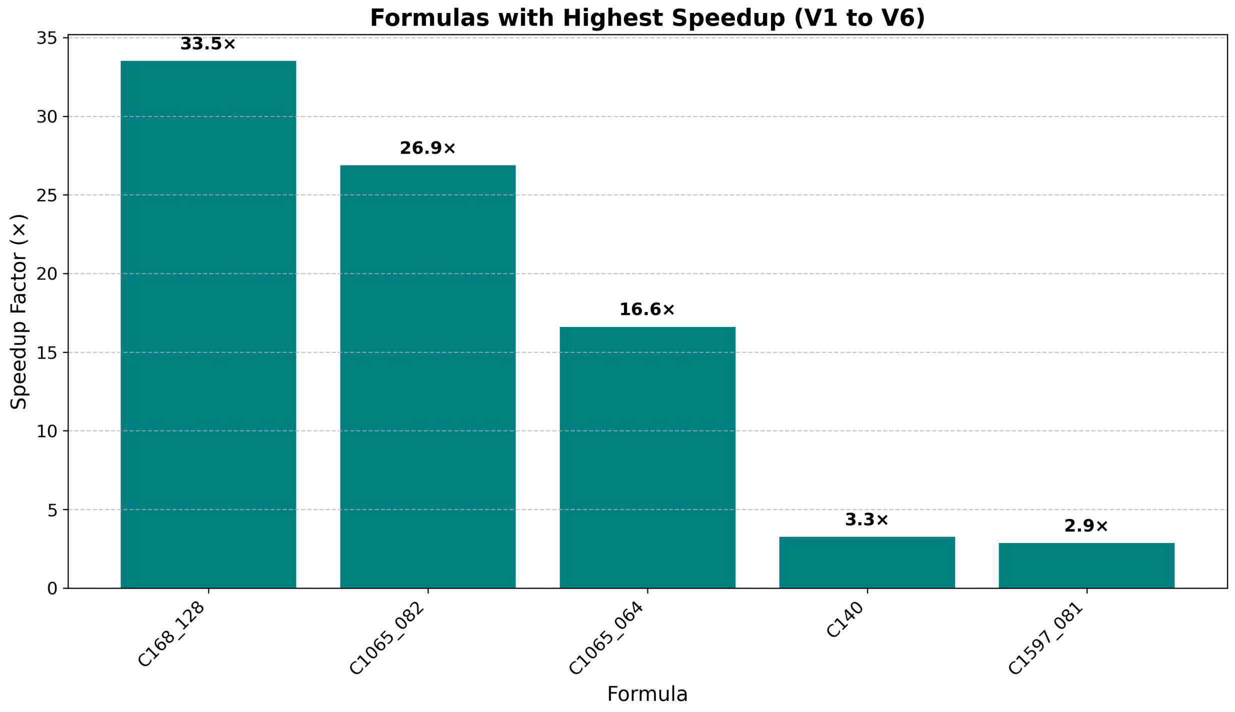

Figure 1: Bar chart displaying the formulas with the highest speedup factors from Version 1 to Version 6. C168_128 shows the most dramatic improvement with a 33.5× speedup.

6 Time Spent

$\sim 20 + 20$ hours