1 Introduction

This is our report for the first Prescriptive Analytics project on SAT solving. Given a CNF in DIMACS standard, our code will attempt to find a satisfying assignment or return UNSAT if there are none. In this report we outline our approach to this project and what steps we took (and why) during our optimization phase. We spent about 8-10 hours each on this project. The contributions are mentioned along with each section.

2 Language Choice

2.1 Choosing Haskell (Mark and Neev)

One of our first attempts at optimization when starting on this project was trying out a lazy language, with the idea that this would help us avoid actually computing branches of the search that would prove to be unnecessary. For this task, we chose to use Haskell. However, we ran into several issues with this approach and eventually switched to using Python instead.

In particular, we encountered difficulty in performing low-level optimizations such as reducing deep copies of our data structures (see section 3.2). Haskell also made it difficult to use several libraries necessary for the SAT solver, particularly randomly choosing branches to switch on and timing the computation.

2.2 Switching to Python (Mark and Neev)

We switched to Python for the project and re-implemented a base SAT solver primarily using dictionaries (hash maps) for keeping track of assignments and standard Python lists. To improve performance, we ran our solver with PyPy, a JIT for Python that sped up our solve times by nearly 50x. We generated our final results on a machine with 8 cores and 32GB of memory.

Note: To ensure that our solver was correct, we took advantage of the fact that it is fast to check if an assignment is valid when our solver gave a satisfying assignment. We wrote a solution checker in both Haskell and Python early on. Sometimes, we needed to use an open-source implementation of MiniSAT to debug our code: https://github.com/master-keying/minisat

3 Optimization Techniques

3.1 Algorithmic Techniques

3.1.1 Heuristics (Mark)

We tried using several search heuristics to help us decide which variable to split on and what assignment to give it. In particular, we implemented randomized (random choice among top five candidates) of MOMS, DLIS, and Jeroslow-Wang. We also tested simply choosing the first variable in our variable set and giving it a random assignment.

We found that changing the heuristic did not make a significant difference in solve time (and, in fact, sometimes slowed it down since the heuristics take linear time to compute), so in our final version we opted to simply choose the first variable and randomly assign it to either true or false.

3.1.2 Random Restarts & Multiple Processes (Mark)

We implemented time-based random restarts (with several solver processes) to help avoid our solver taking too long on a particular instance. To do this, we spin up solver processes separate from the main process. Then, after a particular timeout, the main process terminates the solver process and restarts it with double the timeout (i.e. expontential back-off). This lets us avoid the heavy-tailed distribution of typical SAT solver runs.

We used 7 solver processes (and one main watcher process) since we ran on an 8-core machine. Our initial solver timeout was 30 seconds, with the timeout doubling every time there was a restart.

3.1.3 Watched Literals (Mark)

We tried implementing watched literals (the implementation can be found on the watchlit branch of the repository we submitted). We did this by extending our Clause class to keep track of two literals in the clause and implement a special "remove literal" method that shuffles those as literals are removed and automatically performs unit clause elimination once the clause only has one literal left.

For the relatively small instances we were given, we found only a slight performance increase, so we opted to not use it in our final submission so that other performance improvements (e.g. deep copy management) would be simpler to implement.

3.1.4 Attempt at CDCL (Neev)

We attemped to use CDCL based methods as seen in recent papers and in the class lectures. Conflict driven clause learning involves identifying conflicts in the inference graph that is built up during search and using those to prune the search process. You end up using non-chronological backtracking to skip over multiple useless paths and are able to eliminate similar patterns during search.

However, we quickly realized that the inference graph data structure and extensive refactoring needed would take too long to achieve given the deadline. We also thought that the overhead involved in using our own unoptimized CDCL framework would inhibit our performance, not help it. In the end, we decided our resources were best spent elsewhere. We decided we would optimize in the traditional sense: reducing copying where possible and improve cache/memory usage to reduce overhead.

3.2 Performance Improvements (Mark)

We focused on reducing the amount of memory we need to store before splitting in our search. A naive implementation would store a copy of the assignment and a copy of the remaining variables before each split on a variable. However, this results in large amounts of memory being taken up by function calls waiting to complete (due to the recursive nature). To avoid this, instead of making deep copies, we simply store the changes made, and if we need to backtrack, we simply restore those changes (since the rest will be unchanged).