Project 1: Mass Customization (SAT solver)

Solution Strategy

We implemented our solution in an incremental manner. All of the implementations are simpler or modified version of the DPLL algorithm.

Naive Implementation

Our naive implementation chooses random branching literal, and only removes unit/pure literals once right after it reads the file. It was only able to solve toy examples.

Heuristics

Jeroslow-Wang, two-sided Jeroslow-Wang, DLCS, and DLIS heuristics were implemented. We experimented with each of these heuristics but none of them solved more than three instances on Leaderboard.

Intermediate unit/pure literal removal

We included unit/pure literal removals for internal nodes on our search tree. Removal is done repeatedly until it does not change the state of the current CNF and unassigned variables. At this point, two-sided Jeroslow-Wang heuristics could solve all but two of the instances on Leaderboard.

Multiprocessing

The solver class uses four processes, each of which uses one of two-sided Jeroslow-Wang, Jeroslow-Wang, DLCS, and DLIS heuristics. The processes are terminated when one of them solves the instance. At this point, all instances on Leaderboard could be solved within 5 minutes.

Multiprocessing with mixed strategy

An additional process, which uses mixed strategy is introduced. On each node on the search tree, a heuristics is sampled randomly from distribution `heurDistribution`. `heurDistribution[0]` is the probability that a branching literal is chosen purely randomly, while `heurDistribution[1]`, `heurDistribution[2]`, `heurDistribution[3]`, and `heurDistribution[4]` are probabilities that a branching literal is chosen using two-sided Jeroslow-Wang, Jeroslow-Wang, DLCS, and DLIS respectively. It is possible to change the distribution or add more deterministic heuristics by modifying lines 30-39 in `src/sat_solver.py`.

Experiments and Results

Experiments were done on department machines. `logs` directory contains all the log files for experiments with different strategies (Note that all log files are results from independent instances of `runAll.sh`):

- Log files starting with `uniform` prefix indicate the results where `heurDistribution` is [0, 0.25, 0.25, 0.25, 0.25]. That is, mixed strategy only chooses deterministic heuristics uniformly.
- Log files starting with `best` prefix indicate which heuristics finished each case first. (Such files can be generated by commenting out line 314, uncommenting lines 13 and 58 of `src/sat_solver.py`, and running `runAll.sh`.)
- Based on the result from `best` log files, `heurDistribution` was chosen. We gave more weights to heuristics that succeeded on more instances and heuristics that succeeded more on instances with greater time to solve.
- Log files starting with `weighted` prefix indicate the results where `heurDistribution` is [0, 0.5, 0.2, 0.2, 0.1].
- Log files starting with `randomAndWeighted` prefix indicate the results where `heurDistribution` is [0.03, 0.47, 0.2, 0.2, 0.1].

For most times, one deterministic strategy outperformed other deterministic strategies and mixed strategy, even though the best deterministic strategy is different for each instance. However, there were some instaces where randomness seemed to make the search more efficient:

- `C1597_081` in `uniform1.log` and `randomAndWeighted1.log`
- `C1597_060` in `weighted1.log` and `weighted2.log`: `C1597_060`
- `U50_4450_035` in `weighted3.log`

The best overall total time was from `weighted1.log`, so it is copied to the root directory as `results.log`. (+ The current strategy in the source code is the one used in `weighted` logs.) Note that all the improved instances were satisfiable. This was somehow expected since solving `UNSAT` instances means that we need to make sure that **all branches** fail; it must be generally harder to find variables that would fail early, if exist.

More ideas

Instead of choosing heuristics identically for all nodes on the search tree, we thought about choosing heuristics adaptively based on current situation. For instance, if lengths of clauses vary a lot, it may be better to apply Jerowslow-Wang instead of DLCS or DLIS. Ensemble of heuristics is another option; if two literals have similar Jerowslow-Wang scores, then we may better consult other heuristics instead of strictly following the one with similar scores.

We could have also implemented switching concurrently between two possibile branches of a variable instead of sequentially traversing one branch and then the other one. Using threads may be helpful here, but we need to make sure that we do not double the number of threads for every depth to avoid concurrency issues like thread contention.

Lastly, we tried to implement random restarts to avoid heavy-tailed behavior, a technique that all of the modern SAT solvers utilize. After implementing it, however, we found that it was difficult to tune the hyperparameters necessary to really make use of the random restarts, and thus we decided to not use it for our final implementation (though the code is in our random_restart branch). We also tried to use a JIT compiler to speed up our Python code, but found it overly difficult to re-write our entire codebase to follow Numba's documentation.

PIIs

(Aaron, awang167, a) and (Junewoo, jpark49, Computer)

Time spent

12 hrs = 3 hrs (naive implementation) + 4 hrs (various strategies) + 4 hrs (experiments) + 1hr (report)