1 csci2951o sat solver

Daniel Cho, dcho24, T1

1.1 strategy

High-level overview on the iterations of the solver:

1. ($\sim$ 30s on C1065_064) basic DPLL,
2. ($\sim$ 20s) 2 watched literals and bitsets,
3. ($\sim$ 10s) vsids variable decisions,
4. ($\sim$ 2s) multi-threading.

The solver still struggles to solve
C175_145, C200_1806, C243_188, C289_179, C53_895,
in time.

1.1.1 data structures

The main data structure for my solver are bits and bitsets. Cpp has it's own implementations, but they are fixed at compile time, meaning we have to hard-code the size regardless if there are 50 or 2000 variables. As a result, I made my own bit and bitset structs.

bit This structure contains a single variable stored as an ULL.

bitset This structure contains more than one variable, and has vector of ULL's for positive and negative bits (each item in the vector can represent multiple bits, but larger variables are stored down the length of the list).

1.1.2 preprocessing

During preprocessing, the solver does a reduction mapping where variables are mapped onto a continuous sequence (avoids skips where sometimes the file won't have a variable).

1.1.3 2 watched literals and bitsets

The solver implements 2 watched literals and bitsets to quickly run unit propagation. The solver used to prune pure literals, but removing this improved performance (process of finding pure literals was slow in a data structure for 2 watched literals).

1.2 vsids

Initially, the solver chose variables completely randomly, but implementing VSIDS increased performance and the number of solved problems. In 2 watched literals, it's very easy to implement and performed the best compared to all other decision heuristics.

1.2.1 multi-threading

I initially had a simple implementation of multi-threading by pre-computing a few branches and having a solver for each branch. If one returned sat, it would stop all workers. But sometimes this would mean some workers find an unsat and just sit still.

A more sophisticated approach that solves this is having a pool of workers that takes assignments off a stack and processes them. Each workers also uses luby restarts so that after some amount of iterations it would stop and add everything it was looking at back to the stack so each worker does not spend too much time on a slow assignment. Then, the worker will randomly pick an assignment off the stack to search over.

1.2.2 other attempted strategies

I tried implementing CDCL, but my implementation was slower than not using it. There are quite a lot of operations in CDCL:

1. trace back through the trail,
2. generate the learned clause,
3. find the backtracking point,
4. ensure compatibility with iterative solving,

which, when conducted for each conflict, added quite a lot to the runtime. Moreover, the memory required in adding clauses (to watched literals), keeping trails, etc. was not negligible. Another issue was multi-threading–because each worker needs to keep track of its own trail, watched literals, etc., there was a lot of overhead if a worker were to switch assignments.

Maximum occurrence of clauses of minimum size (MOMS) was a decent heuristic. In cases when the current solver took a long time, introducing MOMS reduced runtime from $\sim 150s \rightarrow \sim 50s$. However, for problems that only took $< 0.1s$, it would increase the runtime to $\sim 100-200s$. I added a few workers with MOMS, but their performance boost in my solver was negligible and sometimes runs up the runtime.

I also tried choosing the variable with the most occurrences, but this did not do well at all for any of the instances.

time commitment: $\sim$72 hrs over 3 weeks