SAT Solver

In implementation of a SAT Solver, we use the DPLL algorithm. The main idea of the algorithm is a brute force searching optimized using clause learning.

3 main components of the algorithm are

- unit propagation
- decision
- learning

Unit Propagation

Unit propagation is a simple piece of deductive system that, given a CNF formula with some clause being a singleton, if the clause were to be satisfiable, then the literal in the clause must be true.

One can recursively apply unit propagation to a formula as many time as needed to simplify a formula down to an easier (or sometimes solved) formula.

There is not much room for logical improvement here. However, optimal implementation is far from naive.

Decision

Decision is the process of selecting the next literal to guess. Depending on the choice of this guess, the runtime of the algorithm varies. Although it is possible to just use arbitrary literal in the decision phase, researches have shown that different heuristics yield different speed up, at least in *general* cases.

Learning

Learning produce clauses based on the Resolution Rule. The process of learning is discussed in the lecture. The idea is that there is a clause that can be added to the formula preserving the original meaning but helps pruning the search tree.

Implementation

Firstly, a straightforward implementation was done. The runtime of the program was too slow due to many reason. Most time was spent in trying to update the value after the assignment for each clause. Initially, the time taken to run a `example/yang/8_UNSAT` was over a minute.

Python based benchmarking and improvement, such as, changing underlying data structure, writing a custom method to copy, using global variables instead of passing arguments, help improving the runtime of the file down to 3 seconds.

However, this implementation uses almost a minute to run `example/yang/6_SAT` and over 5 minutes to run `example/yang/7_UNSAT` after all possible non-major optimization on data structure.

By using a profiler to measure the problem, it was found that the function that assign the clause with the assignment was called a lot of times, and take a lot of time to run each call. Since there is no optimization that reduces the time taken, this leads to implementation 2.

Optimization: 2WL

Instead of assigning the value to each clause, it is possible to do so lazily. This technique is called the 2WL, or 2 Watch Literals. The idea is that instead of assigning each clause, it is possible to keep a watch of only 2 literals per clause. Now, when an assignment is made, it is enough to only check for the two watch literals, and only determine the actual clause if one of the two literals is assigned.

This way, the implementation will be a lot faster for satisfiable formulas with large clauses as it allow skipping most of the calculations in those clauses. This implementation takes 20 seconds on `example/yang/6_SAT` but still uses a little more than first implementation for `example/yang/7_UNSAT`, probably due to a larger overhead.

What 2WL give is the laziness. Especially when the clauses are big, the solver can skip most calculation on each clause, meanwhile, the watching clause is of size 2 except only when the underlying formula is of size 1 or less. This invariants help avoiding calculation when only some information regarding the clauses are need.

Moreover, in this implementation, the clauses are not evaluate (at all), but lazily deduce and adjust itself every time a new information has arrived. That is, when some literal is known to be assigned, the clause remove that literal (or remove itself from the formula). This way, the variable `formula` keeps over-approximating the actual formula when assigned. This open the possibility of using *pure-literal propagation* later on.

However, this speed is only due to luck. It is the case that these example files can be solved using this decision maker in a hundred conflicts, which is not generally the case. The decision maker was well suited with these instances, but would not be successful generally. For example, running on example from sat lib with only 403 clauses takes 20 seconds since it generated 2000 additional conflict clauses.

Heuristics

From the start, the heuristic used was *taking the first literal seen in the formula that is not assigned*. It turned out that this heuristic, although seems arbitrary and inefficient, was in fact efficient for the given test cases. This is because it (randomly happen to) matches the *MOM* (to be discussed) heuristic for the given examples.

Upon more investigation, it was discovered that the reason the previous naive implementation was great is because it happens to select the clauses that are small first. This idea existed and is called *MOM*, short for *Maximum Occurrence in Minimal clause*. As the name suggests, it chooses the maximum occurred literal from the smallest clause. This greedy algorithm were then implemented, yielding a mere 6 seconds runtime on the `example/yang/6_SAT` but still could not manage to get the `example/yang/7_SAT` down under a minute.

The idea of *VSIDS* and the *Berkmin* heuristic was found during literature review. The implementation of these two heuristics, that sources claim efficient, are not very complicated.

The idea of the both heuristics are similar. VSIDS tracks the number of present variable by adding when a learned clause is added to the formula, then periodically divide by 2 to favors recent clauses more than odd clauses. This means that decision will prefer clauses with high activity, which is desirable.

The Berkmin heuristic was proved efficient during the test run. The given test cases could be ran under one and a half minutes. It is possible to optimize the heuristic further, but I plan to explore another idea first. This bring me to the next point

I ended up using MOM and Berkmin together by opting for more MOM earlier on and exponentially switching to Berkmin.

Pure Literal Elimination

Pure literal elimination is the idea of assigning all literal which occur pure, that is only occur positively or negatively, form the formula. While this implementation allow doing a pure literal elimination, it did not became successful since pure literal elimination takes time but did not significantly reduce the search space. While it reduces the number of clauses, the clauses were already optimized by 2WL, so it could not manage to significantly speed up those things. This phenomena was also explained here. However, a pure literal elimination is used during the formula preprocessing.

Restart

The idea of restarting a search was also found in this review and this article. The idea of restarting a search is very unintuitive since restarting a search means the progress will be reset. However, this is not true since a restart preserves learned clause, which is the progress of the search. A restart is effective due to the nature of these programs that are based on luck. One search could take ages and another search could be very quick. Restarting allow trying multiple paths so that the answer will be found expectedly quicker.

Forgetting a clause is also mentioned in many resources, including the one above. Forgetting a useless clause can help speed up the calculation since the total number of clause decreases. Knowing which clause to delete is not trivial. In this implementation, a simple heuristic of *least recently used* is employed

References

references is linked in orange