Project 1: Mass Customization

Sean Segal & Ansel Vahle
Screen name: mario

October 8, 2017

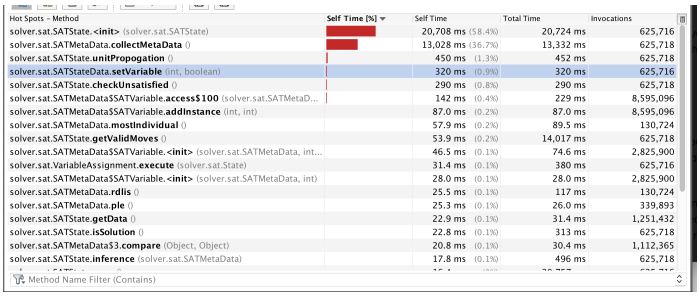
Solution Strategy

We began this project by basing the underlying structure on the generic search method from the discussion from class. We started with the simplest search approach - we initially implemented a basic SAT solver that simply branched on each variable in order, assigning it first to "True" and then "False" until the problem was found to be SAT or UNSAT. As expected, this only worked on the smallest of instances.

From here, we implemented Unit Propagation, which offered a significant speed up. We began keeping track of what clauses we had and had not satisfied, storing the indexes of the unsatisfied clauses as a `Set<Integer>`. Now, when we satisfied a clause, we would remove its index from the set and then we would only search the unsatisfied clauses for unassigned variables. The next step was implementing Pure Literal Elimination. Once again, we only iterated over unsatisifed clauses to improve performance.

At this point, our SAT solver was successful with some of the easier instances, like `solveable.cnf`, but but was still unable to finish on the larger instances. At this point, we sought to eliminate the naive branching from our solver and implemented a search heuristic. We intitally created a SATMetaData class that iterated over the unsatisfied clauses and kept track of the counts of each instance and the clause lengths. Then, using this information we decided how to branch - giving a trial run for each of the search heuristics and then eventually settling on Jeroslow-Wang with a randomization of the top three results. Attempting to utilize the Metdata more effectively, we changed the Pure Literal Elimination logic to rely on it.

At this point, we had implemented all of the components of a standard DPLL algorithm and yet we still weren't able to solve most of the instances. We were combing through our code attempting to find any kind of optimization. However, given the complex branching structure of this kind of problem, the naked eye was unable to tell where our program was spending most of its time. We finally decided to use the VisualVM profiler to test where most of the time was spent running our code. We noticed that most of the time was spent copying our state from one move to another.



Hot Spots - Method	Self Time [%]	Self Time	Total Time	Invocations
solver.sat.SATState.**<init>** (solver.sat.SATState)		20,708 ms (58.4%)	20,724 ms	625,716
solver.sat.SATMetaData.**collectMetaData** ()		13,028 ms (36.7%)	13,332 ms	625,718
solver.sat.SATState.**unitPropogation** ()		450 ms (1.3%)	452 ms	625,718
solver.sat.SATStateData.**setVariable** (int, boolean)		320 ms (0.9%)	320 ms	625,716
solver.sat.SATState.**checkUnsatisfied** ()		290 ms (0.8%)	290 ms	625,718
solver.sat.SATMetaData$SATVariable.**access$100** (solver.sat.SATMetaD...		142 ms (0.4%)	229 ms	8,595,096
solver.sat.SATMetaData$SATVariable.**addInstance** (int, int)		87.0 ms (0.2%)	87.0 ms	8,595,096
solver.sat.SATMetaData.**mostIndividual** ()		57.9 ms (0.2%)	89.5 ms	130,724
solver.sat.SATState.**getValidMoves** ()		53.9 ms (0.2%)	14,017 ms	625,718
solver.sat.SATMetaData$SATVariable.**<init>** (solver.sat.SATMetaData, int...		46.5 ms (0.1%)	74.6 ms	2,825,900
solver.sat.VariableAssignment.**execute** (solver.sat.State)		31.4 ms (0.1%)	380 ms	625,716
solver.sat.SATMetaData$SATVariable.**<init>** (solver.sat.SATMetaData, int)		28.0 ms (0.1%)	28.0 ms	2,825,900
solver.sat.SATMetaData.**rdlis** ()		25.5 ms (0.1%)	117 ms	130,724
solver.sat.SATMetaData.**ple** ()		25.3 ms (0.1%)	26.0 ms	339,893
solver.sat.SATState.**getData** ()		22.9 ms (0.1%)	31.4 ms	1,251,432
solver.sat.SATState.**isSolution** ()		22.8 ms (0.1%)	313 ms	625,718
solver.sat.SATMetaData$3.**compare** (Object, Object)		20.8 ms (0.1%)	30.4 ms	1,112,365
solver.sat.SATState.**inference** (solver.sat.SATMetaData)		17.8 ms (0.1%)	496 ms	625,718

Method Name Filter (Contains)

After coming to terms with this information, we completely changed our underlying representation of the problem. Rather than copying the HashMap from state to state, we decided to add an undo method to the `Move` class which effectively made them invertible. Next, we rewrote the Search algorithm recursively so that we could very easily backtrack by making a call to the `execute` function before the recursive call and the `undo` function after the recursive call. Furthermore, we stopped computing the MetaData and

instead started relying on collecting the information at every state. These changes offered significant speed ups, allowing us to complete several more instances and improved our ties on the existing ones. From here, we decided to add an `isConflict` method to the State so that we could stop branching immediately once we found a conflict.

At this point, we could solve all but one instance. As a last ditch effort to solve our final instance, we added watched literals, which significantly reduced our time on some instances, but increased the time on others which made us fear that there was a bug in updated the watched literals when backtracking. Our final realization was that when computing the Jerslow-Wang scores for each variable, we were using the clause initial sizes rather than the sizes of the clauses once assigned variables were deleted. After this small change, we saw an incredible improvement in our times which allowed us to solve all instances. At first, we chose randomly from the top 3 scores but there was too much variance (we saw some instances vary from 0.88 seconds to over 100 seconds) and so we decided to remove the randomness for our submission. We realize that randomized restarts would probably be a better approach in the future.

To summarize, our SAT solver changed significantly from when we began implementing it. In the end, it relied on a branching data structure that was built for making backtracking easier. We utilized unit propagation, pure literal elimination, and Jeroslow-Wang. Please see the next section for our reasoning behind these choices and other observations we had throughout this process.

Observations

Much of our initial experimentation came with the inclusion of different inference methods and changing the search heuristic. There were obvious gains from using unit propagation at first and then with adding pure literal elimination. Likewise, as we began using more and more advanced search heuristics, our solutions on average began increasing, but in certain cases MOMS might work better Bohm's. As a result of this, we chose to rely on the Jeroslow-Wang algorithm as it consistently boasted the best times. Even while using the Jeroslow-Wang algorithm. At first, we simply relied on the initial clause size and determined where to branch using that information, fearing that we would lose too much time determining the actual clause size. When we switched to using the actual information, our times dropped across the board and we were able to solve all of the instances. We were able to improve our best times by adding a bit of entropy in the mix and choosing at random one of the top three solutions.

Bearing this in mind, our largest gains came from how we chose to represent the problem rather than how we attempted to solve it. With our first iteration, we focused too heavily on maintaining a strong representation of the problem at every level. We were passing down a copy of the HashMap of variable assignments, the counts of the literals of the variables that were unassigned, and a HashMap containing the indices of the unsatisfied clauses. There was a significant overhead not only for the amount of memory being used, but also for the sheer amount of time it took to continuously write and rewrite this information.

When we eventually accepted this and approached the problem from the beginning, we sought to more closely imitate our visualization of the problem and our approach when solving it by hand. We focused on creating a branching data structure that made branching and backtracking much simpler and less data intensive. By eliminating these overheads, we were able to spend more of our computation time on actively deriving the most likely solution. In the end, we relied on a representation that kept track of the move after it and if it failed, then we simply undid the move at that moment. Rather than passing down pertinent information, we calculated it just in time and kept track of information that would be helpful in the future, e.g. if there were several units that could be propagated.

We spent a total of 40 hours on this project.