Screen Name: SwissZombie jigglypuff_jam
Name: Anderson Addo, Ocean Pak
CS Login: aaddo, cpak4

Brainstorming Stage:
During the brainstorming stage, we considered a variety of different ideas. We quickly identified that conflict driven clause learning would be the most successful strategy, but due to the massive implementation effort, we decided to just stick with DPLL and find other ways to optimize performance.

We chose Go as our programming language because it was pretty fast compared to Python and Java. We were also more familiar with the language since we worked together before in previous classes.

An optimization idea that we had was to implement multithreading when selecting a random variable during the recursive backtracking stage. However, due to time constraints, we had ultimately abandoned this idea.

Another idea that we had was to use a stack for every state-changing operation. For every operation such as variable assignment and literal deletion from a clause, we were thinking of storing into a stack, and popping operations off the stack if we (and undoing them) if we had to backtrack. However, due to time constraints, we never pursued this idea. Moreover, while we would be trading space for time massively, we were not sure if the overhead of undoing many operations upon backtracking would be better than the overhead of copying states (such that in backtracking we just throw away the UNSAT state).

Initial Solution Strategy:
As it was highlighted in class, we quickly identified watched literals as an important optimization. However, we were not quite sure how to implement watched literals in a way that minimizes as much restoration work as possible. However, we still were able to use watched literals as a way to separate the solver's state (e.g. what variables were assigned and what clauses were modified) and the original formula, which helped limit the amount of copying in the program. Regardless, our design uses a decent amount of map assignments and copies.

Optimizations for Final Solution Strategy:
After we got an initial working solution, we were still pretty unsatisfied with the performance. It was always accurate, but it could only solve C140.cnf in /input (though it could solve other smaller SAT instances, such as the toy inputs). As such, we added in 2 optimizations:

1. We added in restarts. After a certain backtrack limit was hit during a run, the run would restart with a slightly shuffled variable branching order (which is the order in which variables are looked at after each round of pure literal and unit clause elimination). The branching order was based on the number of literal appearances per variable in the SAT solution; it was slightly shuffled each restart to add in randomness. After a certain number of restarts, the backtrack limit would increase by a constant. We noticed that this made it faster at solving C140.cnf, but not much else.
    a. Once we finished both optimizations, we surmised that waiting a few restarts before increasing the backtrack limit is more beneficial when the limit is lower, since each restart takes less time. So, we added in logic such that every time a backtrack limit is

eventually increased, the number of restarts before the next increase is reduced and the increment the backtrack limit will receive will increase.

2. The branching order was initially shuffled a bit and then kept constant for the whole search. The order (albeit shuffled) was based on variable instance appearance count. Of course, once you get deep into the search, that order will no longer reflect the current state of the SAT formula. So we added in logic to re-create a new branching order once you get to certain depths.
    a. This order *can* be ranked based on variable instance appearance count. Another way of ranking is to use a heuristic that ranks variables by how pure they are (purity scores < 0) at that state of the SAT formula. From our empirical results, we *think* purity scores performed better, and as such chose to use it in our final solution strategy.

Experimental Observations:
Even after those two optimizations, our solution was still rather lackluster. In order to determine what was causing significant time bottlenecks, we used Go's builtin benchmarking tools to help visualize the stack trace. A flame graph that we generated for our program can be seen in Appendix A.

From the flame graph and memory utilization stats we figured that state copying was our main bottleneck. Luckily, we found a way to reduce the number of copies by half. By realizing that whenever you branch on a variable, no matter how deep you are in the search, you only need one copy of the original state, not two. The copy is used to test one assignment, and the original is used to test the negated assignment (and if neither works we backtrack and throw everything away, so nothing is polluted). This significantly improved performance and allowed us to solve multiple SAT instances in /input within the 5 minute time limit. However, we still cannot solve the instances with more than 1000 clauses.

Another observation that we found with our program was that we were able to solve SAT instances a lot quicker than UNSAT instances. This makes sense, as there are a lot more cases to cover in order to determine the UNSAT case, which would take more time.

Note on Time: We approximately spent around 25 hours on this project, mainly due to solving implementation bugs.

Appendix A:

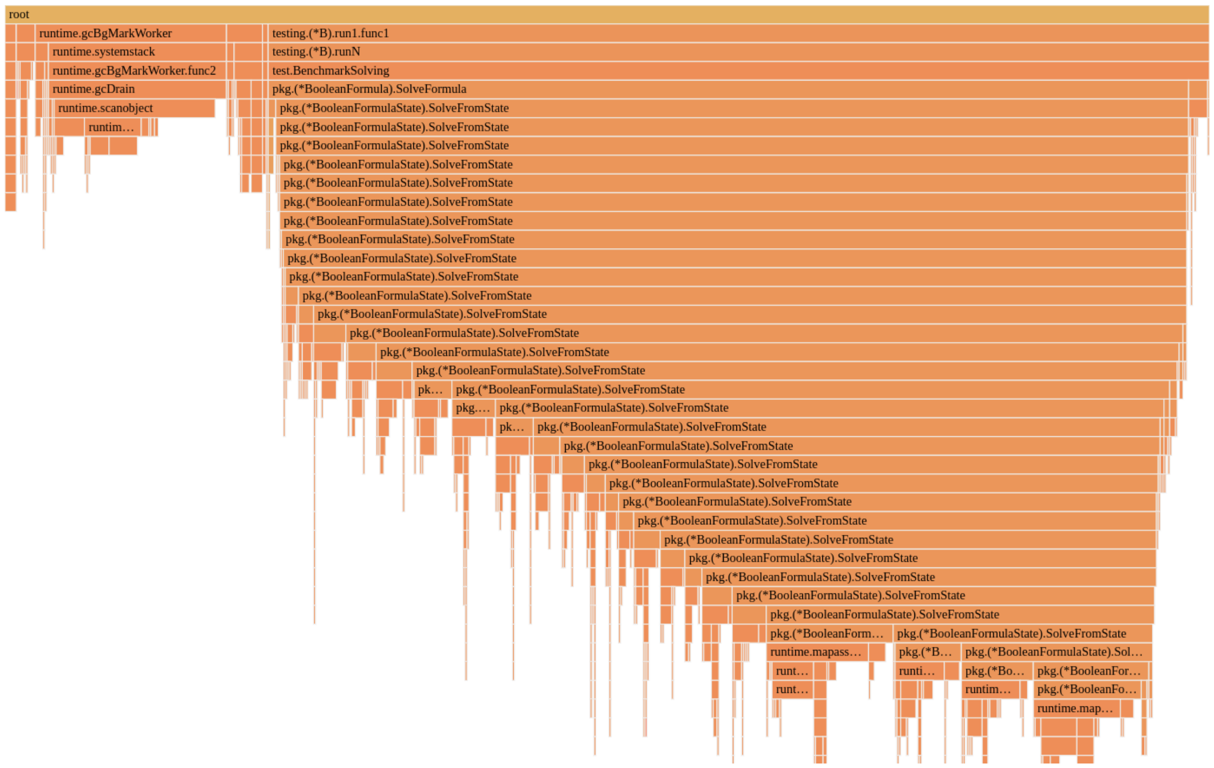

Figure 1: Flame Graph generated by Go Benchmarking tool

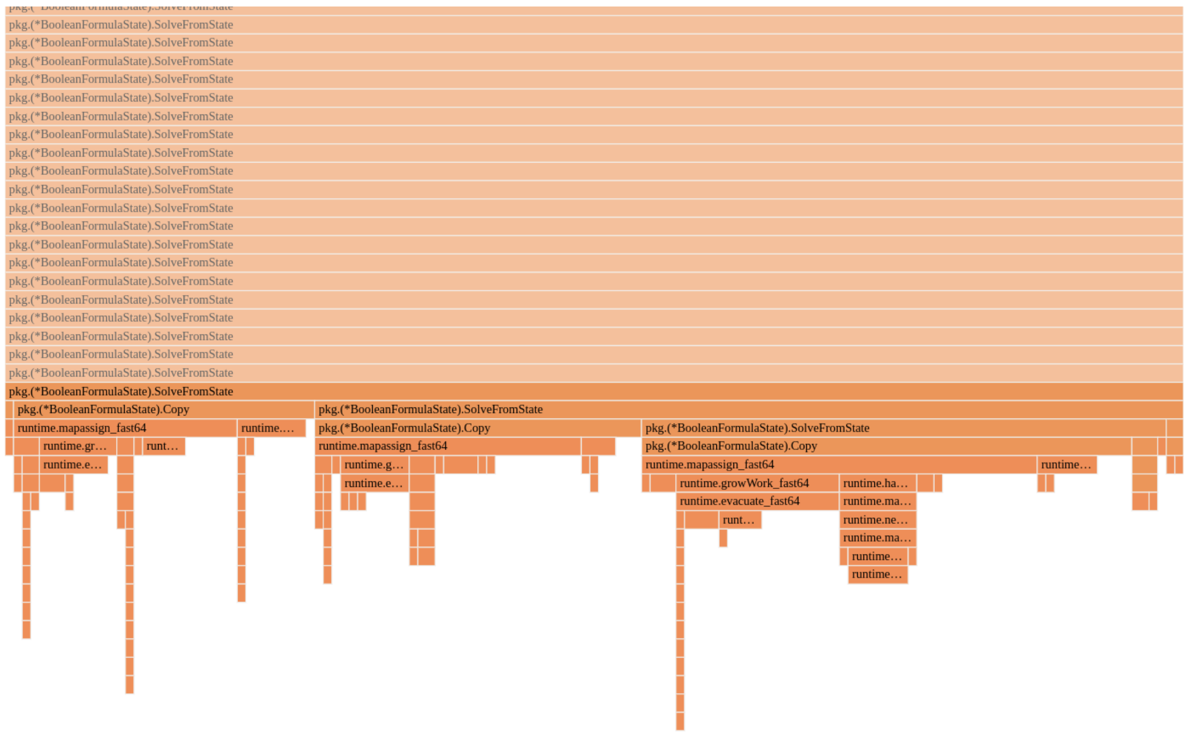

Figure 2: Close up of function calls in flame graph - notice the expensive copies