1 Solution Strategy

We implemented an iterative version of DPLL (a backtracking tree-search algorithm). Our solution makes use of interesting heuristics and efficient data structures on top of optimizations to the original DPLL pseudocode from lecture. We wrote a checker to confirm that assignments made by our solver for SAT instances did, in fact, satisfy the instance. Furthermore, we used a pre-existing SAT-solving library (`pysat`) to verify that the UNSAT instances were truly unsatisfiable.

While we tried several other approaches, we found that we got the best results from our C++ implementation of an iterative version of DPLL, which used a custom branching heuristic we developed called "ClauseReducer". You can find more information about our implementation in the DPLL Optimizations and Heuristics sections.

2 Heuristics

We read papers about branching heuristics and came across a lot of interesting ideas, many of which were from this paper by Lagoudakis and Littman. Some of these were **MAXO** (maximally occurring variable), **MOMS** (variable that occurs the most in clauses of minimal size), **JW** (Jeroslaw-Wang heuristic), and **MAMS** (picks the variable with the best combined score from MAXO and MOMS). Also, the paper mentioned **SUP**, a look-ahead method that gets variables from aforementioned heuristics, and picks the one that creates the most unit clauses.

Within the SUP heuristic, we made a tracker (which has since been removed) to see which heuristics were selecting variables that were ultimately being picked by SUP. We saw that MAXO and MOMS were being picked most often, with JW and MAMS collectively being chosen less than 20% of the time in most instances. Thus, we ran SUP with only MAXO and MOMS to save time.

We also wrote a heuristic **DeepSUP**, which checks the unit propagations caused after propagating the initially created unit propagations (deep version of SUP), up to a certain depth in the future. However, we found this performed worse than ClauseReducer (below), so we didn't implement it again when we shifted from Java to C++.

After a conversation with Serdar, we made a heuristic that got the literals from MAXO, MOMS, JW, and MAMS (or some subset of these) and picked randomly between them (saves time since we don't have to count the number of unit propagations created, as in SUP or DeepSUP). However, this didn't perform as well as SUP.

Eventually, we implemented our own heuristic **ClauseReducer (CR)**, which is a modification of SUP. It gets the variables picked by MAXO and MOMS and compares them based on how many unit propagations the MAXO variable would create in the positive branch, and how many unit propagations the MOMS variable would create in the negative branch. This increases the chance that we will have a good outcome in both branches of the variable (positive and negative literal). Overall, we found that we got the best results with this heuristic.

	MAXO	MOMS	JW	MAMS	SUP	CR	CR (C++)
C1065_064.cnf	73.01	14.75	19.58	53.04	20.33	9.62	4.23
C1597_060.cnf	78.26	237.98	–	–	–	8.41	2.67
C1597_081.cnf	–	–	127.22	–	104.14	45.47	20.77
U75_1597_024.cnf	221.04	–	12.23	53.36	12.89	1.24	0.13

Table 1: The table compares the times (in seconds) for four instances using different heuristics in Java (except for the last column which is implemented in C++).

3 DPLL Optimizations

- **Efficiently find pure literals**: We use vectors to store the counts of positive and negative literals (`positiveLiteralCounts[i]` stores the number of occurrences of literal $(i+1)$, and `negativeLiteralCounts[i]` stores the number of occurrences of literal $-(i+1)$). Using these, as we add/remove literals from clauses, we can check if a new pure literal is created in constant time (without looping through clauses). It also makes computing MAXO faster.
- **Efficiently find unit clauses**: When we unit propagate a literal, in clauses where its negation is found, if the new clause size is 1 we track it as a unit clause. Thus, we can find unit clauses as they are formed in constant time (without looping through clauses).
- **Backtrack with stack**: We use stacks and global variables for backtracking. We globally store indices of remaining clauses, literals removed from each clause, and model assignments, so we don't have to mutate the instance. Our stack stores edits we make to the global variables at each branch and makes unrolling changes during backtracking efficient.
- **Iterative DPLL**: Rather than the recursive definition of DPLL from lecture, we implemented an iterative version. We used a stack to keep track of the variable we branch on, and whether or not the negative branch has been checked yet. Getting rid of the function-call overhead from recursion made significant improvements locally, but not on the department machine.
- **Language**: C++ gave us a 31% speedup over our original Java implementation.

4 What Didn't Work

- Originally, we made a copy of our state at each branch. This made backtracking trivial, but profiling revealed that copying took $\sim 30\%$ of our runtime. We switched to a faster method of using stacks to backtrack without copying state.
- We used a TreeMap (sorted map) to map counts to the variables that occur that many times. This made finding the MAXO variable an $O(log\ n)$ operation (where n is the number of variables). However, profiling showed that maintaining the TreeMap (keeping the keys sorted) was costly, and our solver was faster without it.
- We tried different combinations of the heuristics used in SUP. Combinations including JW and/or MAMS and combinations excluding MAXO and/or MOMS made the solver slower.
- To optimize branching variable selection, we implemented a version of DeepSUP that would only run on the n shortest clauses in the instance. This made our solver slightly faster when choosing between MOMS, MAMS, MAXO, and JW, but we ultimately removed this because we only kept MOMS and MAXO (which doesn't go through all of the clauses anyways).
- We tried a version of DeepSUP which used an adaptive lookahead depth (based on the number of variables and clauses), however, no configuration we tried performed better than CR.

5 Time Spent

15 hours per week.