For this project, I considered two different implementations: a naïve DPLL implementation with different branching strategies, and a more involved CDCL implementation with watched literals and VSIDS heuristic. Both solutions were implemented in C++. As a submission, I decided to include the former one, since its performance on the given inputs was better. Over three weeks, I spent approximately 50 hours on project development and literature review.

DPLL Solver

Data Structures & Tracked Metadata

In this implementation, a formula was represented by a vector of pointers to clauses and a vector of pointers to variables. In turn, clauses and variables were represented by two mutually dependent structs.

The clause struct tracked whether a given clause was active (in other words, not yet satisfied), and if not, by which variable it was satisfied (helps a lot when backtracking), the number of active literals in the clause, and a vector of pointers to the variables representing all literals present in the clause (both active and not).

The variable struct tracked the numeric ID of the variable (in order to make it printable), its currently assigned value (T, F, or Undefined), and two vectors of pointers to its positive and negative occurrences.

This layout allowed accessing variables from clauses and vice versa by pointer dereferencing, which decreased the time spent on propagation. Importantly, the references to variables and clauses were set up once and for the rest of the program, so propagation and backtracking were reduced to updating the counts and variable values to avoid the cost associated with inserting and removing values from vectors.

Methods of the Formula Class

- `find_unit_clause` – iterates over all clauses and returns the first one that is active and has only one active literal (both values are from the metadata);
- `find_pure_literal` – iterates over all variables and returns the first one that is active and has either no negative occurrences or no positive occurrences (determined from the metadata).
- `propagate` – deactivates the variable, iterates over all positive and negative occurrences, and either marks clauses as satisfied or decrements the number of active literals in the clause. Most importantly: propagate calls itself recursively, once decrementing the number of active literals in the clause yields a unit clause (this means that we only need to call `find_unit_clause` only in the beginning – all other propagations are automatic). It also returns all propagated literals to be passed in `depropagate` once backtracking.
- `depropagate` – does the opposite of `propagate`, operates on the list of propagated variables.
- `solve` – main function, also recursive. Maintains the stack of current search choices. On the first invocation, calls `find_unit_clause` and propagates those literals. Then, eliminates pure literals, checks whether the formula became SAT or UNSAT (by iterating over all clauses), branching on it recursively, and depropagating if the result of the branch is UNSAT or returning the propagated variables if the result is SAT.

Finally, the heuristic used for branching is Jeroslow-Wang, since it has shown superb results on the input data. All other heuristics from the slides were implemented as well. More on that in the evaluation.

CDCL Solver

Data Structures & Tracked Metadata

In this case, the metadata was more rich. In the same fashion, a formula was represented by a list of pointers to clauses and a list of pointers to variables, and a list of branching choices, which was now stored on a per-formula

basis. Furthermore, the number of newly encountered conflicts and the current restart ceiling were stored as well. Now, a conflict occurrence counter was added to each variable, while a clause representation was shrunk to a list of literals and a pair of watched variables.

Brief Description of Solving

Instead of giving a detailed description of every method, I will briefly describe an overall approach to solving. On each iteration of the solution function, depending on the current state, random restarts (strategy from Luby et al.) and VSIDS decay is invoked.

Then, a function iterating over all clauses is called, checking the watched literals and trying to reassign them in case some of them are not active. The same function triggers propagation or conflict analysis for the appropriate clauses, and once those are done, branches using the VSIDS strategy.

Propagation reassigns the watched literals in the satisfied clauses and calls back the watched-literals-checking function to check the clauses in which the variable could be still present (overestimated by the list of positive/negative occurrences, both active and not), in case assigning the variable led to some of the clauses being unsatisfiable or unit.

Conflict analysis iterates over the list of previous choices, trying to find the variable with the conflicting assignment. Then, since the incoming edges of the implication graph are stored along each choice (easy to deduce, since we know in which clause the variable became unit), the contrapositive of those is added as a clause, and the number of conflicts for each variable is updated. Then, a non-chronological backtrack is called, iterating over the list of choices in reverse order and finding the closest one present in the conflict clause.

Evaluation

On the input data given, the DPLL solver performed much better than the CDCL solver. It took less than 60 seconds for the DPLL solver to solve all formulas, while its counterpart exceeded the 5-minute limit on all formulas with more than 1000 clauses in them. There are a couple of reasons why it might have happened:

1. It seems that the Jeroslow-Wang heuristic could be just a lucky choice for the data at hand, since other heuristics (DLCS, DLIS, MOMS, etc.) perform much worse on the testing data. This conjecture could be checked by running the solution on a wider pool of possibly random examples.
2. The CDCL solution generates conflicts at a much slower rate than similar solvers (such as Minisat). While I have not looked at the differences in architecture (I only used Minisat as a performance reference), profiling shows that the vast majority of the running time ($> 90\%$) is spent on checking watched literals and performing propagation, which could happen for either of the reasons below:
   (a) VSIDS heuristic is a poor choice for the input data, so too many propagations are happening because of bad branching decisions. This could be evaluated by using different branching strategies or tweaking the decay parameter of VSIDS.
   (b) Conflict analysis does not look far enough in the implication graph, while looking further could allow for registering conflict clauses with fewer literals (or even finding unique implication points). This could be evaluated by tweaking clause analysis.
   (c) There is a bug in either conflict analysis or propagation, so that more than necessary clauses are being looked at. However, the solver is still producing correct, even if slow, results.

While I would love to check all of those conjectures, due to the limited time allocated for the project, I am leaving those just as suggestions.

To conclude, I enjoyed the project a lot and feel more confident in my understanding of SAT-solving. As a person who learns best by doing, implementing the strategies proved crucial for my understanding of those.