PROJECT-1: MASS CUSTOMIZATION
2/28/2025

Evan Meyer	emeyer7	Spugizakom
Lexi Henrion	ehenrion	Athena

IDEAS:

Initially we thought about representing each clause as a linked list of literals. The idea is that, within each DPLL instance, there is a pointer for the effective start of the clause, and any literal prior to that pointer is to be thought of as not being in the clause. When a literal is to be removed from a clause, it is simply swapped with the literal at the current effective start of the clause, and then the start pointer is incremented by 1. We quickly realized that this idea could be achieved even more easily by representing each clause as an ordinary list, and by using indices as the pointers.

Here is an example, with the literals at the start indices in bold:

Original clauses: [[**3**,8,1,4,6], [**6**,2,7,9], [**9**,7,8,4,1], [**7**,6,4,3]].
Original start indices: [0,0,0,0].
Set 6 to false.
Modified clauses: [[6,**8**,1,4,3], [6,**2**,7,9], [**9**,7,8,4,1], [6,**7**,4,3]].
Modified start indices: [1,1,0,1].

That representation has the advantage that only lists of start indices would need to be copied, updated, and passed between functions, rather than lists of sets.

However, in the input CNF files, most of the clauses are very small, and therefore working with lists of start indices instead of lists of clauses actually wouldn't save much time or computational effort. For this reason we decided that the list-of-lists structure would just be adding unnecessary complexity to the code.

We ended up taking the hint from the java support code and decided to represent the clauses as a list of sets. This makes sense because sets don't contain duplicates, and any literal duplicated within a clause from a CNF formula has no effect on the evaluation of the clause.

We additionally decided to remove clauses from the list which contained a variable with both of its polarities, because such clauses will evaluate to true regardless of variable assignments. This was achieved in the initialization step by simply not appending such clauses to the list of clauses being accumulated.

Another idea we had was to maintain a global list-of-lists-of-clauses, and only append the modified versions of clauses to the clauses that had actually been modified.

Here is an example, using the same clauses as the example above:

Original clauses: [[{3,8,1,4,6}], [{6,2,7,9}], [{9,7,8,4,1}], [{7,6,4,3}]].
Set 6 to false.
Modified clauses: [[{3,8,1,4,6},{3,8,1,4}], [{6,2,7,9},{2,7,9}], [{9,7,8,4,1}], [{7,6,4,3},{7,4,3}]].

In this idea, appending is used rather than replacement, to allow backtracking. In the example, there is only one clause which doesn't get modified, but in actual instances of DPLL, there are often many clauses which aren't modified, meaning that a lot of time and computational effort needed to copy those clauses could be avoided. However, we decided not to go with this idea for a similar reason to why we abandoned the list-of-lists idea, namely because lists of indices of modified clauses would still need to be maintained.

Pure literal elimination is a relatively time intensive step. Therefore, we chose to only remove one level of pure variables within each instance of the pure literal elimination step, rather than repeatedly checking to see if any new pure literals had been created by the removal. Also, part of our heuristic was to decide when to or not to perform pure literal elimination at all, and we did this by keeping a running number of how many values have been removed since the most recent instance of pure literal elimination.

To select the next branch at each step, we decided to simply pick a random clause. If the clause is non-empty, then its first value is taken as the next branch variable. But if it's empty, then a random variable is simply selected from the unassigned variables set. This strategy ensures that there is a slight bias towards selecting a commonly-occurring variable rather than an uncommonly-occurring variable, allowing the solver to reduce the problem more quickly.

OBSERVATIONS:
In the python implementation, using shallow copies of the clauses list ended up creating quick incorrect UNSAT declarations. After opting instead for using deepcopy, however, the program slowed down significantly. This is because, under the hood, deepcopy is designed for general use, but in this case a simple list comprehension of set copies is much more optimal, so the implementation uses that instead. Also, at every stage, there are two sub-branches, copies only need to be made for the first sub-branch which is checked, since if the second sub-branch fails too then it doesn't matter that the current version of the clauses list was modified. We were comfortable with different languages so we have both Java and Python solvers. This was helpful early on for playing with implementation and checking work off of each other. Our initial naive Java DPLL solver solves toy_feasible in 0.20 seconds, the smarter Java solver solves it in 0.02 seconds, and the python smart solution (our final solution) solves it in 0.0012509822845458984 seconds.

ALGORITHM:
In our implementation, the DPLL function takes in the next branch variable as input.
In the main program, the entry into the recursive solver is done by calling DPLL on 0.
Since 0 is not a value found in the clauses, nothing will happen in this initial branch execution step, but initial unit propagation and initial pure literal elimination will happen if available.

```
dpll(b):
        Branch execution step:
                Add b to the assignments, clear sets with b, and remove (-b) from sets.
        Unit propagation step:
                Repeat:
                        Find unit values to remove:
                                Accumulate a set of unit values, and if a conflict is found then indicate
                                this and break out of the repetition.
                        If no unit values were found, break out of the repetition.
                        Remove the accumulated unit values:
                                Pop unit values from the unit values set, add them to the assignments,
                                and remove them from the clauses, until the unit values set is empty.
        Pure literal elimination step:
                        Find pure values to remove:
                                Accumulate a set of traversed values and non-pure values, by
                                adding all variables to the traversed set, and by only adding values to
                                the non-pure set which are present in both polarities. Compute the pure
                                values set as the difference between the traversed and non-pure sets.
                        Remove the accumulated pure values:
                                Keep popping pure values from the pure values set, adding them
                                to assignments, and clearing the clauses in which they occur, until
                                the pure values set is empty.
        Next branch selection step:
                If no variables are remaining to be assigned, then indicate SAT.
                Otherwise, take a value from a random clause if that clause is non-empty, and remove
                the variable from the unassigned variables set, but if the clause is empty, then simply
                pop a value from the unassigned variables set. Call dpll on the variable with a copy of the
                unassigned variables set and a copy of the clauses list. If this substep indicates SAT,
                then update the assignments set and indicate SAT. Otherwise, call dpll on the negative of
                the variable, with references to the unassigned variables set and to the clauses list
                (because there is no need to use copies at this stage). If this substep indicates SAT, then
                update the assignments set and indicate SAT. Otherwise, indicate UNSAT.
```