Kent Nguyen (A0093934W)
itaqmail@gmail.com

CS 4244 PROJECT: CDCL SAT SOLVER
FINAL REPORT: STAGE 1

1. SOLVER'S ARCHITECTURE

The project is written in Java 8. The solver `CdclSolver` contains 4 smaller decoupled modules:

- `FormulaPreprocessor` (optional): Performs initial processing on the formula, such as eliminating pure literals (i.e. variables having only one polarity throughout the whole formula).
- `UnitPropagator`: Applies unit clause rule and assigns values to unit literals.
- `BranchPicker`: Selects a variable and value to assign.
- `ConflictAnalyzer`: Analyzes the most recent conflict and produces a new clause learned from the conflict.

This separation of concerns allows great flexibility during testing, where a module can be replaced or disabled easily. Usage of the solver can be found in the `Main` class:

```
SatSolver solver = new CdclSolver()
        .with(new PureLiteralElimination())
        .with(new HybridVsidsPicker())
        .with(new NaiveUnitPropagator())
        .with(new ClauseLearningWithUip());

Assignment assignment = solver.solve(formula);
```

The `Formula` object represents a CNF formula, and contains a conjunction of `Clause`s, which each is a disjunction of `Literal`s. Each `Literal` is a pair of an integer `variable` (a number from 1 to N, the number of variables) and a boolean `isNegated`.

The `Assignment` object holds a map of `SingleAssignment`s for all variables, each of which stores necessary properties such as the variable's value, decision level and antecedent, as defined in *The Handbook of Satisfiability*. `Assignment` also keeps track of the current decision level, and the kappa antecedent. The submodules aforementioned will manipulate and modify `Assignment` accordingly.

`BranchPicker`'S IMPLEMENTATIONS

The following implementations are provided:

- `InteractivePicker`: Lets the user enter the assignment to pick, purely for debugging purposes.
- `LinearPicker`: Chooses the smallest index unassigned variable (x_1, x_2 …). This is also used mainly for debugging, where determinism is needed.
- `RandomPicker`: Chooses a variable randomly from the current pool of unassigned variables.
- `VsidsPicker`: This closely follows the Variable State Independent Decaying Sum (VSIDS) heuristic as described in Chaff (M. Moskewicz, 2001), with a slight variation: instead of always choosing the first unassigned literal with the highest score, the literals after it can be chosen at a configurable probability. This is to prevent the same set of literals from being picked too often, slowing down the process. Also, all scores will decay by half after 256 learned clauses.

- `HybridVsidsPicker`: This combines both `VsidsPicker` and `RandomPicker`, allowing the latter to take place 10% of the time (also configurable).
- `BootstrapPicker`: This wraps around another `BranchPicker`, and bootstraps the assignment with a list of predefined literals, before giving back the responsibility to the wrapped picker. Used mainly for debugging purposes.

`ConflictAnalyzer`'s Implementations

This module has one implementation `ClauseLearningWithUip`, which is a faithful interpretation of the conflict analysis description in *The Handbook of Satisfiability*. It also detects Unit Implication Points (UIPs) and stops learning until the first UIP is identified.

An improved version of the conflict analyzer will be discussed in ***Stage 2*** of the report.

`UnitPropagator`'s Implementations

At first, the implementation `NaiveUnitPropagator` was used. After it was found out that the solver spent most of its time during propagation, the more efficient `TwoWatchedLiteralPropagator` was implemented.

Refer to Section 5, "**`UnitPropagator` choices**" for a more thorough performance analysis.

Backtracking

Backtracking is performed inside `CdclSolver`, after receiving the newly learned clause from `ConflictAnalyzer`.

The second highest decision level among the learned clause's literals is chosen as the new decision level. If all literals are on the same level (with the current conflicting decision level), by convention the backtracked decision level is set to zero. Otherwise, if the learned clause is empty, -1 is returned and will effectively stop the solving process.

When the new decision level is determined, all assignments with higher levels will be removed.

2. SAT Formulas Generation

CNF formulas can be randomly generated by the method `FormulaHelper.generateCnf`, the outputs of which are written to external files. Subsequently, these files can be parsed by `FormulaHelper.parseFromFile` into `Formula` instances.

The `generateCnf` method receives 3 parameters: number of variables `N`, number of literals per clause `K`, and number of clauses `L`. For the scope of this project, `K` is fixed to 3. Intuitively, it has been observed that the higher the ratio `L/N`, the higher probability that the formula is unsatisfiable.

A majority of the unsatisfiable formulas generated by `FormulaHelper` are found to be "easy", i.e. the solver takes insignificant amounts of time to solve. This is possibly because a totally random generator tends to generate clauses that are similar, or even logically redundant.

Some researching leads to other existing generators that are guaranteed to generate "harder" formulas. For example, the tool `CNFgen` "encodes structured combinatorial problems well known to be challenging for certain SAT solvers" (Lauria, n.d.), while `ToughSAT` encodes other difficult problems such as Factoring, Subset Sum and Random k-SAT (Bebel, n.d.).

3. Correctness of Implementation

The solver `CdclSolver` produces a non-null assignment if the formula is satisfiable. The original `Formula` instance is then evaluated with the returned `Assignment` to check whether the algorithm is correct.

In the case that the formula is unsatisfiable, the solver returns null. In **Stage 2** of the project, a prover is implemented to generate refutation proofs, which can be verified and thus confirms the solver's correctness.

4. Performance Findings

The performance was benchmarked by the elapsed time taken by the solver as a whole, the elapsed time taken by the `UnitPropagator` module alone, and the invocation count for `BranchPicker`.

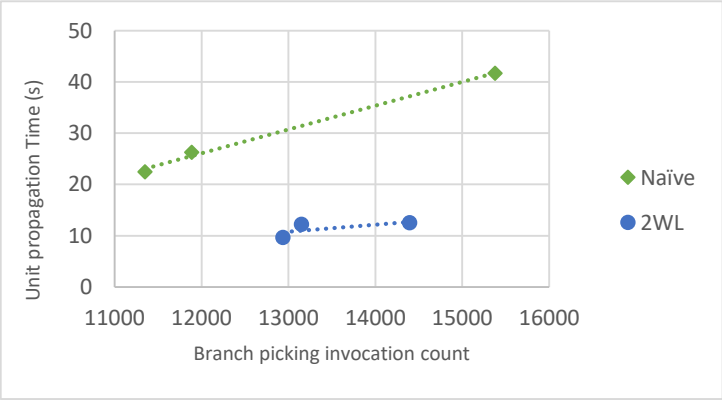
`BranchPicker` Choices

At first, `LinearPicker` and `RandomPicker` were used to test simple CNF formulas. These performed similarly fast for variable count $N \leq 50$. However, both started to struggle for larger and harder formulas. For *`/others/60var_unsat_hard.cnf`*, `RandomPicker` needed about 4s and 7000 branches picked, while `LinearPicker` failed to solve it in reasonable time. It was noted that the time spent on unit propagation shot up as well, taking a considerable portion (50-90%) of total compute time.

Then, `VsidsPicker` was tested with formulas containing 100 and above variables. As expected, the solving time reduced significantly for many test cases. For example, `VsidsPicker` only needed about 1s (on my laptop) to find a satisfying assignment for the formula in *`N150_K3_L650_sat.cnf`*, while the previous pickers were too slow for the timing to be reported.

For several cases, `HybridVsidsPicker` performs considerably faster than `VsidsPicker`, for instance the former peaked at less than 1s to solve the 16-queen problem (*`/others/queens.cnf`*), while `VsidsPicker` constantly spent around >15s. This is probably because the formula contains mostly 2-clauses, and about half of the literals share similar scores as well as similar chances to be found in learned clauses, which means the same literals will be picked most of the time. Thus, `HybridVsidsPicker` adds some needed stochasticity to the branch picking process, allowing the solver to jump out of exponentially harder branches.

`UnitPropagator` Choices



No matter which `BranchPicker` was used, it was observed that the solver spent most of its time performing unit propagation. Therefore, the popular two-watched-literal (2WL) heuristic was implemented and benchmarked against the default `NaiveUnitPropagator`. Due to the stochastic nature of `HybridVsidsPicker`, the measured time varied greatly, thus the branch picking invocation count was also taken under consideration.

Above is the plot of the **time spent on unit propagation** against the **branch picking invocation count** of the solver with input from *`N150_K3_L650_unsat_hard.cnf`*:

It can be seen that the `NaiveUnitPropagator` is twice as slow compared to `TwoWatchedLiteralPropagator`. This is clearly because `NaiveUnitPropagator` attempts to scan all literals in the formula (which gets increasingly larger as more clauses are learned), while the other only keeps track of two literals from each clause, and only assigning values to these watched literals would trigger the propagation process.

OTHER CONSIDERATIONS

The solver also supports "pure literal elimination" (available as a preprocess module), however it seems to have little effect on hard contradictory formulas, which usually have very few pure literals (both polarities are present), and also generate a large amount of new clauses during conflict analysis, rendering any initial reduction of pure literals insignificant.

For several larger test cases, the solver still takes a long amount of time to deduce satisfiability, even with `HybridVsidsPicker` and `TwoWatchedLiteralPropagator`. Finer measurements showed that the solver now spent most of its time on conflict analysis. The below outputs is obtained after factoring a large number (*`/others/factor_sat.cnf`*):

```
Total time: 153889 ms
Unit propagation time: 18205 ms
Branch picking invocation count: 24442
Branch picking time: 3408 ms
Conflict analysis time: 128938 ms
Final formula size: 21333
```

As seen above, branch picking took only 10% of the time, while conflict analysis took 80% of the time. At first glance, this might be explained by the large number of clauses generated by `ClauseLearningWithUip` (originally from 7585 to a whopping 21333).

Nevertheless, a deeper analysis of the current clause learning heuristic pointed to a more subtle problem, which led to an improvement on the algorithm that results in a significant performance boost. This improvement will be discussed in more details in ***Stage 2*** of the report.

On the other hand, current state-of-the-art SAT solvers have adopted various advanced techniques to further reduce solving time, such as clause deletion strategies (to reduce the number of clauses), or random restarts (to prevent from getting stuck in an exponentially hard search subtree). These are out of scope of this project.

5. EINSTEIN'S PUZZLE

The CNF formula was generated as guided in (Yurichev, n.d.), and can be found in *`others/einstein_puzzle.cnf`*.

Yellow	Blue	Red	Green	White
Norwegian	Dane	British	**German**	Sweden
Water	Tea	Milk	Coffee	Beer
Dunhill	Blends	Pall Mall	Prince	Blue Master
Cat	Horse	Birds	**Fish**	Dog

Result: The German owns the fish.

6. LESSONS LEARNED

After the first stage of this project, I have gained a deeper understanding of SAT solvers, and more specifically CDCL solvers. I have also understood how mathematical problems such as the Pigeonhole Principle and Einstein's Problem can be encoded in CNF formula. Most importantly, I've had a lot of fun implementing various heuristics, including those considered to be cutting edge such as VSIDS and 2-watched literals.

Lastly, I'm looking forward to improve my SAT solver with more modern techniques to solve harder formulas more quickly.

SELF-ASSIGNED GRADE: **A**.

7. REFERENCES

Bebel, H. Y. (n.d.). *Tough SAT Project*. Retrieved from https://toughsat.appspot.com/

Lauria, M. (n.d.). *CNFgen - Combinatorial benchmarks for SAT solvers*. Retrieved from http://massimolauria.github.io/cnfgen/

M. Moskewicz, C. M. (2001). Chaff: Engineering an Efficient SAT Solver. *39th Design Automation Conference (DAC 2001)*.

Yurichev, D. (n.d.). *Zebra puzzle as a SAT problem*. Retrieved from Dennis Yurichev's Blog: https://yurichev.com/blog/zebra_SAT/