Kent Nguyen (A0093934W)
itaqmail@gmail.com

CS 4244 PROJECT: CDCL SAT SOLVER
FINAL REPORT: STAGE 2

1. FROM SOLVER TO PROVER

In **Stage 1** of the project, a CDCL SAT solver was implemented, which could generate assignments for satisfiable formulas. However, when the solver stated that the formula is unsatisfiable, it was unknown whether the statement was truly correct.

This report will aim to overcome this shortcoming by discussing how an unsatisfiability prover can be implemented (and the solver modified) to generate refutation proofs for unsatisfiable formulas. These proofs can be easily verified, thus confirming the correctness of the solver.

During the process of implementing the refutation prover, a problem was found in the solving algorithm that turned out to be the underlying cause of the long solving time for several hard unsatisfiable CNF instances, as discussed in **Stage 1** (e.g. the formula in `/others/factor_sat.cnf` took the solver more than 2.5 minutes). As a result, an improvement to the clause learning heuristics described in *The Handbook of Satisfiability* is proposed, which helped to significantly boost the solver's performance. This will be discussed in more details in the section "**Identifying the Culprit**" below.

2. TRACING RESOLUTIONS

In theory, CDCL SAT solvers and their predecessor, the DPLL algorithm, can be viewed as resolution-based proof systems, i.e. they can readily produce refutation proofs for unsatisfiable formulas (P. Beame, 2004).

In the case of CDCL solvers, clause learning is in fact a repeated application of the resolution rule. The clause learning procedure (with 1-UIP) from *The Handbook of Satisfiability* is reproduced below for reference:

$$\omega_L^{d,i} = \begin{cases} \alpha(\kappa), & \text{if } i = 0 \\ \omega_L^{d,i-1} \odot \alpha(l), & \text{if } i \neq 0, l \in \omega_L^{d,i-1}, \delta(l) = d, \alpha(l) \neq \text{NIL} \\ \omega_L^{d,i-1}, & \text{if } \omega \text{ has only one literal assigned at level } d \neq 0 \end{cases} \quad (1)$$

where κ represents the conflicting node in the implication graph, $\alpha(l)$ denotes the antecedent clause of a literal node l (thus $\alpha(\kappa)$ refers to the unsatisfied clause), $\delta(l)$ is the decision level of the literal l in the current assignment, and thus $d = \delta(\kappa)$. Also, let $\omega_L^{d,i}$, with $i = 0,1, \ldots$, be the intermediate clause obtained after i resolution operations, and $\odot$ be the resolution operator.

Therefore, formula (1) can be interpreted as follows: upon a conflicting clause $\omega = \alpha(\kappa) = \omega_\kappa$, it is repeatedly resolved with the antecedent clauses of ω's literals that are on the same conflicting level d, until no more resolution can be done (or ω contains only one literal on level d, for 1-UIP). In short:

$$\omega_{learned} = \left(\left((\omega_1 \odot \omega_2) \odot \omega_3 \right) \ldots \odot \omega_n \right) \quad (2)$$

where $\omega_2, \omega_2, \ldots, \omega_n$ are the clauses that the conflicting clause $\omega_1 = \omega_\kappa$ is resolved with, and $\omega_{learned}$ is the clause to be learned by the solver. If we use this information, we can draw the resolution derivation tree of the learned clause $\omega_{learned}$, which is shown on the right.

$$\dfrac{\dfrac{\dfrac{\omega_1 \quad \omega_2}{\omega_3} \quad \omega_4}{\ldots} \quad \omega_n}{\omega_{learned}}$$

Now, we define the list $[\omega_1, \omega_2, \dots, \omega_n]$ to be the *trace* of the learned clause $\omega_{learned}$, and we can draw the resolution derivation tree more succinctly as below:

$$\frac{\omega_1 \quad \omega_2 \quad \dots \quad \omega_n}{\omega_{learned}}, \text{ which is equivalent to (2)}$$

Note that resolutions still take place from left to right, and we only consider trace of length 2 and above (trace of length 1 means $\omega_{learned} = \omega_1$, which is a case of a redundant learned clause). In addition, only learned clauses have non-empty traces – the formula's initial clauses have empty traces.

It is also clear that all clauses $\omega_1, \omega_2, \dots, \omega_n$ in the trace of $\omega_{learned}$ should have already existed/been learned before $\omega_{learned}$ is appended to the formula.

In the case of an unsatisfiable formula, the resolution process would produce an empty clause, i.e. a bottom $\perp$ clause. If we treat this special clause as the terminal clause of the formula, we can draw a similar resolution derivation tree for it:

$$\frac{\omega_1 \quad \omega_2 \quad \dots \quad \omega_n}{\perp}$$

If any of the input clauses $\omega_1, \omega_2, \dots, \omega_n$ have non-empty traces, we can expand them as sub-derivation trees. For example, if ω_2 has a trace, we'd expand it as follows:

$$\frac{\omega_1 \quad \dfrac{\omega_{21} \quad \omega_{22} \quad \dots}{\omega_2} \quad \dots \quad \omega_n}{\perp}$$

Repeating the process, we now have produced a refutation proof.

3. Implementing the Prover

Modifications to `CdclSolver`

The `Clause` class is augmented with a list `trace`, which stores the IDs of the clauses used in resolution to produce it.

In `ClauseLearningWithUip`, at the beginning the `learnedClause` ω is initialized with the value of the conflicting clause, which is also added to the initially empty trace of ω. During clause learning, whenever an antecedent clause is used in resolution, it will also be added to the trace of ω.

When a formula is unsatisfiable and an empty/bottom clause is produced, `CdclSolver` now assigns the bottom clause with its trace to the `Formula` instance before returning the `null` assignment.

Now the formula contains enough information for `UnsatProver` to generate a refutation proof.

`UnsatProver`'s Implementation

The class `UnsatProver` is to be initialized with a formula containing a bottom clause. Its `prove` method would produce a `Proof` instance, which contains a list of `Clause`s and a list of `Resolution`s. `Proof` also defines methods to export in `String` format or write in text files, as well as a method to verify itself by testing all resolutions.

The `prove` method performs a breadth-first search starting from the bottom clause and explore all clauses in its trace, before moving on to repeatedly explore the traces of the inner clauses. This process would find all clauses needed for the proof, and for those clauses that have non-empty traces, it would also generate the corresponding `Resolution`s.

A `Resolution` stores the ID of the output clause, and a list of input clause IDs as the trace. The traces in `Resolution`s produced by `UnsatProver` can have length $n \geq 2$. Since the project description requires 2-resolutions (atomic trace of length 2), the `Proof` class defines methods to expand its n-resolutions (trace of length n) to 2-resolutions, and to renumber the resolved clauses after n-resolution expansion. An example of a *correct* proof generation process is shown below:

```
w1 (1 v 2)
w2 (1 v ¬2)
w3 (¬1 v ¬2)
w4 (¬1 v 2)

Resolutions:

w5 (¬1) <- [w4, w3]
w6 ()  <- [w2, w1, w5]
```

n-resolution expansion →

```
w1 (1 v 2)
w2 (1 v ¬2)
w3 (¬1 v ¬2)
w4 (¬1 v 2)

Resolutions:

w5 (¬1) <- [w4, w3]
w7 (1) <- [w2, w1]
w6 () <- [w7, w5]
```

clause renumbering →

```
w1 (1 v 2)
w2 (1 v ¬2)
w3 (¬1 v ¬2)
w4 (¬1 v 2)

Resolutions:

w5 (¬1) <- [w3, w4]
w6 (1) <- [w1, w2]
w7 () <- [w5, w6]
```

After expansion, the 3-resolution `[w2, w1, w5] -> w6` is broken into two atomic 2-resolutions: `[w2, w1] -> w7` and `[w7, w5] -> w6`, and the resolution for `w7` is inserted between `w5` and `w6`. Then, we restore the numerical order of the clauses by switching `w6` and `w7`. Finally, since the resolution operation is commutative for atomic 2-resolutions, we can sort the two clauses in the trace, e.g. from `[w4, w3]` to `[w3, w4]`.

A Case of Clause Duplication

Nevertheless, for the (obviously) unsatisfiable formula (*`simple/unsat_simple_N3.cnf`*):

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \\ \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \quad (3)$$

early attempts of the prover produced the following (weird-looking) proof:

```
w1 (1 v 2 v 3)        w3 (1 v ¬2 v 3)       w5 (¬1 v 2 v 3)       w7 (¬1 v ¬2 v 3)
w2 (1 v 2 v ¬3)       w4 (1 v ¬2 v ¬3)      w6 (¬1 v 2 v ¬3)      w8 (¬1 v ¬2 v ¬3)

w9  (¬1 v ¬2) <- [w8, w7]
w10 (¬1)      <- [w6, w9, w5, w9]
w11 (1 v ¬2)  <- [w4, w3]
w12 ()        <- [w2, w10, w11, w1, w10, w11, w10]
```

It was first observed that some resolutions contain duplicates of clauses in the traces, and albeit correct, those resolutions are rather inefficient. For example, `w10` can be produced from the shorter trace `[w6, w5, w9]`, while the final bottom clause `w12` can instead be derived from `[w2, w1, w11, w10]`.

IDENTIFYING THE CULPRIT

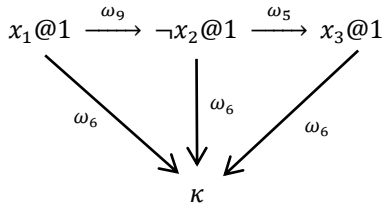
After adding in more detailed logging, it was possible to have a closer look at how resolutions were performed and this suspicious block of logs was found:

```
Learned clause w9 = (¬1 v ¬2)
...
Conflict at clause w6! kappa = (¬1 v 2 v ¬3)

Assignment after propagation: x1@1, ¬x2@1 (w9), x3@1 (w5)
>> Resolving with clause 9 (¬1 v ¬2) => (¬1 v ¬3)
>> Resolving with clause 5 (¬1 v 2 v 3) => (¬1 v 2)
>> Resolving with clause 9 (¬1 v ¬2) => (¬1)

Learned clause = (¬1)
```

Let's first draw the implication graph for the assignment:



From the graph, x_1 is picked, then x_2 and x_3 are propagated, thus all three variables are assigned on the same decision level 1. According to formula (1), we are to resolve ω_6 with ω_5 and ω_9, as both ω_5 and ω_9 are on the same decision level ($d = 1$) with ω_6. However, in what order should we resolve them? *The Handbook of Satisfiability* does not specify this.

If we resolve ω_6 with ω_9 before ω_5 (as x_2 appears first in the clause, which the solver is currently doing), after the two resolutions we receive $\neg x_1 \vee x_2$, and since $\alpha(x_2) = \omega_9 = (\neg x_1 \vee \neg x_2) \neq \mathrm{NIL}$, we have to resolve *again* with ω_9 to get $\neg x_1$.

Meanwhile, if we resolve ω_6 with ω_5 before ω_9, we can directly achieve $\neg x_1$. Intuitively, because x_3 is propagated after x_2, it makes more sense to resolve backwards with $\alpha(x_3) = \omega_5$ first.

We can see that the order of resolving clauses on the same level (due to literals propagated from a branch) makes a difference, therefore it is necessary to look beyond *The Handbook* for a definitive answer.

4. THE IMPROVED HEURISTIC OF CLAUSE LEARNING

I came across an obscured unpublished book draft by Christoph Weidenbach (Weidenbach) that gave a detailed formalization of the CDCL heuristics, which is again mentioned in the paper that I referred to while implementing the two-watched-literal scheme in **Stage 1** (Mathias Fleury, 2018).

It defines a CDCL problem state as a five-tuple $(M; N; U; k; D)$, where M is the current assignment (called a literal *trail* in the paper), N is the set of initial clauses, U is the set of learned clauses, k is the current decision level and D is the current conflicting clause (or top $\top$ if there is no conflicting clause yet).

Consider a clause $(l \vee C)$ in our clause database $N \cup U$, where l is a literal and C is the disjunction of the remaining literals in the clause. If C is logically false due to the assignment M, i.e. $M \vDash \neg C$, and $l \notin M$, the rule of propagation states that:

Propagate $(M; N; U; k; \top) \xrightarrow{CDCL} (M \oplus l^{l \vee C}; N; U; k; \top)$

Here, the literal l is annotated with the clause $l \vee C$ that forced l to become true (i.e. the antecedent of l), before being appended to the assignment M. Note that the assignment order is important, so we use the append operator $\oplus$.

The conflict analysis process is encoded by the following rule:

Resolve $(M \oplus l^{l \vee C}; N; U; k; D \vee \neg l) \xrightarrow{CDCL} (M; N; U; k; D \vee C)$,
where both the level of l and the maximal level of a literal in D is k (to support resolution termination at 1-UIP).

We can see that the conflicting clause $D \vee \neg l$ is resolved against the antecedent $l \vee C$ of the last literal assigned l (end of trail M) to yield the intermediate clause $D \vee C$. This confirms our intuition that we need to resolve in descending order of assignment.

IMPLEMENTING THE CORRECT HEURISTIC

Fortunately, we can support this new heuristic with ease by augmenting a variable x_i with one more property: order, which simply takes its value from a global counter that is incremented whenever a variable is assigned. In other words, the order is stronger than the decision level as it imposes an ordering on those literals propagated on the same level.

Then, in our `ClauseLearningWithUip.analyze`, we sort the learned clause's literals descendingly by their orders before resolutions:

```
// Sorts by descending order
List<SingleAssignment> sortedSingles = learnedClause.stream()
        .map(assignment::getSingle) // Converts each literal to a SingleAssignment
        .sorted((s1, s2) -> Integer.compare(s2.order, s1.order))
        .collect(Collectors.toList());
```

As a result, we obtain a better and shorter proof for formula (3) as shown below:

```
w1 (1 v 2 v 3)        w3 (1 v ¬2 v 3)       w5 (¬1 v 2 v 3)       w7 (¬1 v ¬2 v 3)
w2 (1 v 2 v ¬3)       w4 (1 v ¬2 v ¬3)      w6 (¬1 v 2 v ¬3)      w8 (¬1 v ¬2 v ¬3)

w9  (¬1 v ¬2) <- [w8, w7]
w10 (¬1)      <- [w6, w5, w9]
w11 (1 v ¬2)  <- [w4, w3]
w12 ()        <- [w2, w1, w11, w10]
```

The expanded and renumbered proof can be found in */proofs/unsat_simple_N3.txt*.

5. A WELCOMED BOOST IN PERFORMANCE

The refutation proof can be showed to be a tree-like directed acrylic graph (Nordström, 2015), thus there should not be any clause duplication in a trace, which is true for our second generated proof above. Naturally, this means that

our previous implementation of clause analysis was highly inefficient, which performed a large number of redundant resolutions to arrive at the same learned clause.

Coincidentally, in **Stage 1** we have also noticed that for several formula instances, the solver spends much of its time during conflict analysis. For example, the satisfiable formula in `/others/factor_sat.cnf` took the solver about 154 seconds, 80% of which (130s) are spent on conflict analysis.

```
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
```

How well does our new and improved conflict analysis heuristic perform with the same formula?

```
Total time: 6262 ms
Unit propagation time: 1590 ms
Branch picking invocation count: 13478
Branch picking time: 1907 ms
Conflict analysis time: 1382 ms
Final formula size: 9739
```

Evidently, we have achieved a total solving time reduction of 95.9% from 154s to a mere 6s, and the time spent on conflict analysis drastically dropped to 1.4s as expected.

The performance improvement is also observed with unsatisfiable formulas. The inefficient solver needed 263.9s to decide the unsatisfiability of the hard formula in `/generated/N200_K3_L1000.cnf`, while the new solver only needs 17.4s (93.3% time reduction):

```
// Before:
Total time: 263913 ms
Unit propagation time: 250714 ms
Branch picking invocation count: 66692
Branch picking time: 1523 ms
Conflict analysis time: 9490 ms
Final formula size: 56266

// After:
Total time: 17472 ms
Unit propagation time: 14101 ms
Branch picking invocation count: 40916
Branch picking time: 1066 ms
Conflict analysis time: 1423 ms
Final formula size: 23073
```

This brings our solver on par with other existing solvers such as `pysat`, even without implementing random restarts, and refutes my initial suspicion that Java was the cause of my solver's early low performance.

6. SUMMARY

After the second stage of this project, I have learned even more about CDCL SAT solvers, and also how to use it to generate refutation proofs of unsatisfiable CNF instances.

It was also exciting to come up with an improvement to *The Handbook of Satisfiability*, more specifically to its clause learning heuristic, which results in an apparent performance boost to my solver's implementation.

Last but not least, the source of the solver has been made public, and can be accessed here:

https://github.com/kentnek/cdcl-sat-solver

***To Prof Kuldeep**: Thank you for the change in the course's curriculum that gave us the wonderful opportunity to build our own CDCL SAT solver, and I hope that you will be as delighted reading this report as I was writing it.* :)

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/

Mathias Fleury, J. C. (2018). A verified SAT solver with watched literals using imperative HOL. *CPP 2018 - Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 158-171 .

Nordström, J. (2015). On the interplay between proof complexity and SAT solving. *ACM SIGLOG News*, 19-44 .

P. Beame, H. K. (2004). Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research 2*, 319-351.

Weidenbach, C. (n.d.). *Conflict Driven Clause Learning (CDCL).* Retrieved from https://www.mpi-inf.mpg.de/fileadmin/inf/rg1/script5ws1617.pdf