COMP 509: Project Final Report

Alexis Le Glaunec (`afl5`)

December 3, 2021

Honor Pledge

On my honor, I have neither given nor received any unauthorized aid on this report.

1 Introduction

This work focuses on the **3-satisfiability problem**, i.e. checking satisfiability over formulas expressed in CNF (Conjunctive Normal Form) with at most 3 disjunctions per clause. This problem is known to be NP-complete and therefore it is difficult to find an efficient algorithm to solve it. Therefore, many algorithms have been implemented to tackle this problem (Resolution, Truth Table Enumeration, etc). One of the most successful algorithm is called **DPLL** [2, 1] and its design is still used in modern SAT solvers. This is the design that I have decided to implement for this project using the language Java.

2 DPLL implementation

2.1 Definition of DPLL

The DPLL procedure for SAT solving is based upon two rules: the *Unit-preference rule* propagates unit clauses that must be true (by definition of CNF formulas) and the *Splitting rule* selects a proposition p based on what we call a heuristic, simplify the formula and call the DPLL algorithm recursively. The SAT solver is capable of reading DIMACS-format files through a custom-made parser and returns a truth assignment if a formula φ is satisfiable or "UNSAT" otherwise.

2.2 Why Java?

I have chosen to implement my SAT solver in Java because I believe Java is a good trade-off between elegant languages, like Python, that do not perform well and languages with better performances, like C++, that are prone to bugs and harder to maintain.

2.3 Recursive vs Iterative implementation

I have implementation two SAT solvers in Java:

- **DPLLRecursive**: a recursive DPLL SAT solver that implements the DPLL algorithm in a recursive way using substitution in formulas to simplify formulas.
- **DPLLIterative**: an iterative DPLL SAT solver using a stack to store the calls to DPLL and storing assigned values in propositions instead of the substitution that is done in the recursive algorithm.

At first, I designed the recursive algorithm that is simplier, more elegant but also significantly slow. I realized that unfortunately only when running experiments so I had to come up with a new design that would allow me to run the experiments more quickly. This is why I have designed an iterative version of the DPLL algorithm using a stack. This is the version that I will present in the rest of the report.

2.4 Implementation choices

2.4.1 Unit Propagation

The main work of my implementation has been to properly design the Unit-propagation method. This method is an extension of the unit-preference rules that propagates unit clauses and finds any conflict. It stops only when there are no more unit clauses or a conflict occured. Compared to the standard Unit-propagation algorithm, this design saves time not calling over and over the same method and instead doing all the unit calls at once.

The core design of this method is about finding the unit clauses. To do so, I chose to construct the *Proposition* with two fields *posClauses* and *negClauses* which retains pointers respectively to the clauses where a proposition p occurs as a positive literal and as a negative literal. The purpose of the *value* field is to know if the proposition has already be assigned, and if so, its value to detect potential conflicts. The complete design is presented below:

```
1 public class Prop implements Form {
2     private final String symbol;
3     private Boolean value = null;
4     private List<Clause> posClauses = new ArrayList<>();
5     private List<Clause> negClauses = new ArrayList<>();
6 }
```

Listing 1: Proposition class design

2.4.2 Backtracking

If any conflict is detected in the Unit Propagation step, we call the *backtrack* method over the clauses deduced by the Unit Propagation and the chosen variable that lead to the conflict and we reset their values and put them back in the *unassigned* list of propositions.

2.4.3 Assignments stack

To save memory, every time a decision is made and has no conflict, we only push the deduced clauses and the newly chosen variable instead of the whole new truth assignment on top of the stack. The trade-off is that if the formula is satisfiable, then at the end of the DPLL algorithm we have to empty the stack to re-create the entire truth assignment. This is done this way:

```
1         while (assignments.size() > 0) {
2             Assignment a = assignments.pop();
3             tau.put(a.choice.a, a.choice.b);
4             tau.putAll(a.tau);
5         }
6         return tau;
```

Listing 2: Retrieving the complete truth assignment that satisfies a formula φ

3 Correctness of the SAT solver

The iterative SAT solver implementation is not as trivial as the recursive implementation. Therefore, we must ascertains that it is correct through examples.

3.1 Oracle class

I have implemented an *Oracle* class whose purpose is to test the iterative and recursive algorithms over the same set of random 3-CNF formulas and check on the results. As a result, they always output the same results and therefore I am pretty confident that my iterative implementation of the SAT solver is correct.

3.2 Einstein's problem

To even more assure the SAT solver's correctness, we express a famous puzzle, the Einstein's puzzle, in propositional logic and compare the solver results with the expected results.

First, we define a list of ordered sets that will help us model the problem:

- $\mathcal{N} = \{Brit, Swede, Dane, Norwegian, German\}$

- $\mathcal{H} = \{White, Green, Yellow, Blue, Red\}$
- $\mathcal{B} = \{Tea, Coffee, Milk, Beer, Water\}$
- $\mathcal{C} = \{PallMall, Dunhill, Blends, Bluemasters, Prince\}$
- $\mathcal{P} = \{Dogs, Birds, Cat, Horse, Fish\}$
- $\mathcal{D} = \{1, 2, 3, 4, 5\}$

Then, we can express atomic propositions for the model:

$$\begin{aligned} \forall i, j \in [1,5], n_{i,j} &= \text{"}\mathcal{N}_i \text{ is in house } \mathcal{D}_j\text{"} \\ h_{i,j} &= \text{"}\mathcal{H}_i \text{ is in house } \mathcal{D}_j\text{"} \\ b_{i,j} &= \text{"}\mathcal{B}_i \text{ is in house } \mathcal{D}_j\text{"} \\ c_{i,j} &= \text{"}\mathcal{C}_i \text{ is in house } \mathcal{D}_j\text{"} \\ p_{i,j} &= \text{"}\mathcal{P}_i \text{ is in house } \mathcal{D}_j\text{"} \end{aligned}$$

From these atomic propositions, we define the following clauses to represent the basic model for Einstein's riddle :

$$C_{\geq 1} = \bigwedge_{x \in \{n,h,b,c,p\}} \left(\bigwedge_{i=1}^{5} \left(\bigvee_{j=1}^{5} x_{i,j}\right)\right) \qquad \text{(Each house has at least one } x\text{)}$$

$$C_{\leq 1} = \bigwedge_{x \in \{n,h,b,c,p\}} \left(\bigwedge_{i,j,k=1/j<k}^{5} (\neg x_{i,j} \vee \neg x_{i,k}) \bigwedge_{i,j,k=1/i<k}^{5} (\neg x_{i,j} \vee \neg x_{k,j})\right) \quad \text{(Each house has at most one } x\text{)}$$

Now, it remains to translate the hints into clauses using the atomic propositions previously defined. There are 4 kinds of assertions in the hints. We define a function for each:

$$sameHouse(x, y) = \bigvee_{j=1}^{5} (x_j \wedge y_j)) \qquad (x \text{ and } y \text{ are in the same house})$$

$$leftHouse(x, y) = \bigvee_{i=2}^{5} ((x_i \wedge y_{i-1})) \qquad (x\text{'s house is on the left of } y\text{'s house})$$

$$atomicHouse(x_i) = x_i \qquad (x \text{ is in house } i)$$

$$livesNextHouse(x, y) = \bigvee_{j=1}^{4} (x_j \wedge x_{j+1}) \bigvee_{j=2}^{5} (x_j \wedge x_{j-1}) \qquad (x\text{'s house is next to } y\text{'s house})$$

"The Brit lives in the red house." $\models\dashv \mathcal{H}_1 = sameHouse(n_1, h_5)$
"The Swede keeps dogs as pets." $\models\dashv \mathcal{H}_2 = sameHouse(n_2, p_1)$
"The Dane drinks tea." $\models\dashv \mathcal{H}_3 = sameHouse(n_3, b_1)$
"The green house is on the left of the white house." $\models\dashv \mathcal{H}_4 = leftHouse(h_1, h_2)$
"The green house's owner drinks coffee." $\models\dashv \mathcal{H}_5 = sameHouse(h_2, b_2)$
"The person who smokes Pall Mall rears birds." $\models\dashv \mathcal{H}_6 = sameHouse(c_1, p_2)$
"The owner of the yellow house smokes Dunhill." $\models\dashv \mathcal{H}_7 = sameHouse(h_3, c_2)$
"The man living in the center house drinks milk." $\models\dashv \mathcal{H}_8 = atomicHouse(b_{3,3})$
"The Norwegian lives in the first house." $\models\dashv \mathcal{H}_9 = atomicHouse(n_{4,1})$
"The man who smokes Blends lives next to the one who keeps cats."
$\models\dashv \mathcal{H}_{10} = livesNextHouse(c_3, p_3)$
"The man who keeps the horse lives next to the man who smokes Dunhill."
$\models\dashv \mathcal{H}_{11} = livesNextHouse(p_4, c_2)$
"The owner who smokes Bluemasters drinks beer." $\models\dashv \mathcal{H}_{12} = sameHouse(c_4, b_4)$
"The German smokes Prince." $\models\dashv \mathcal{H}_{13} = sameHouse(n_5, c_5)$
"The Norwegian lives next to the blue house." $\models\dashv \mathcal{H}_{14} = livesNextHouse(n_4, h_4)$
"The man who smokes Blends has a neighbor who drinks water." $\models\dashv \mathcal{H}_{15} = livesNextHouse(c_3, b_5)$

All hints $\mathcal{H}_i$ are expressed in DNF, therefore we consider their transformation in CNF (using the $toCNF$ function of my DNF class) as input for our DPLL solver.

We obtain the following formula that represents Einstein's riddle that we give as entry to the parser:

$$\varphi = \mathcal{C}_{\geq 1} \wedge \mathcal{C}_{\leq 1} \bigwedge_{i=1}^{15} toCNF(\mathcal{H}_i)$$

The output from the solver (executable with the *main* function in *EinsteinPuzzle* class) is as follow:

```
Dunhill is in house 1
Bluemasters is in house 5
Coffee is in house 4
Beer is in house 5
Blue is in house 2
Green is in house 4
Norwegian is in house 1
Horse is in house 2
Swede is in house 5
```

```
Birds is in house 3
Water is in house 1
Milk is in house 3
Blends is in house 2
Prince is in house 4
Tea is in house 2
Red is in house 3
Yellow is in house 1
Pall Mall is in house 3
White is in house 5
Cat is in house 1
Dane is in house 2
German is in house 4
Brit is in house 3
Fish is in house 4
Dogs is in house 5
```

After checking the riddle, this solution validates all the constrains and I therefore consider it as correct. We can know answer Einstein's riddle: as the fish is in house 4 which is owned by the German, we can deduce that **the German owns the fish**.

This validates the SAT solver correctness, as the problem is considered difficult enough to spot many potential bugs.

3.3 Satisfiability of random formulas

The satisfiability curve of 3-CNF random formulas is well-known [3] and can help us verify on a larger scale that our implementation is correct by comparing the probability curve with respect to the $\frac{L}{N}$ ratio of clauses per variable to other SAT solvers curves as it should be independent from the implementation. In figure 1, we see that almost all 3-CNF random formulas are satisfiable until the value $\frac{L}{N} \approx 4.0$. This is because there are only a few constraints per variable represented by the clauses. However, when the ratio $\frac{L}{N}$ is higher than 4.8, almost all formulas become unsatisfiable because there are too many constraints, i.e. clauses, per variable to make them satisfiable. There is a slight difference between $N = 100$ and $N = 150$, as it seems that for $N = 150$, the *satisfiability threshold* [3] is more around 4.6 than 4.8. This is actually close to the results of other SAT solvers [3], although the satisfiability threshold should be closer to 4.25. I suppose that the difference is due to the fact that we are experimenting on a relatively low number of variables (100 and 150). The difference between 100 and 150 variables comes from the fact that $N = 150$ is more accurate.

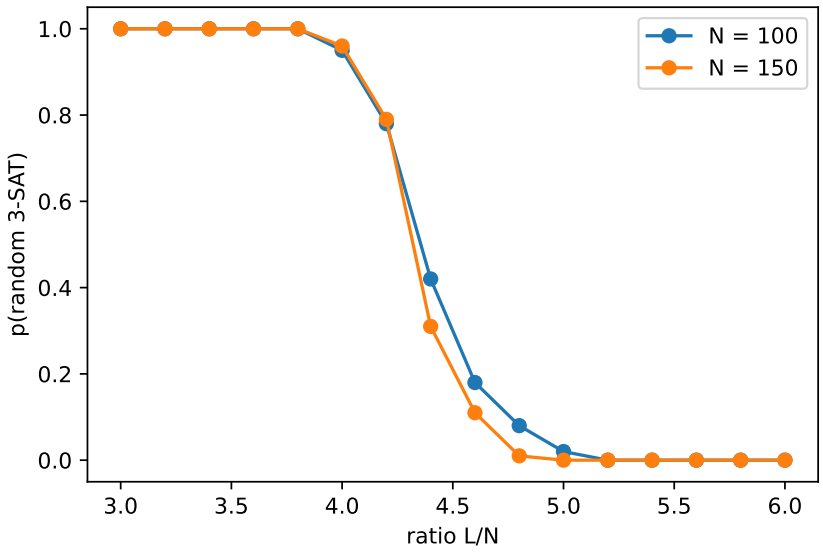

Figure 1: Probability of satisfiability for random 3-CNF formulas for 100 and 150 variables for different ratios L/N of clauses per variable

4 Evaluation

4.1 Random model

We define a random generator class called *randomGenerator* that generates random 3-CNF formulas where every clause is made of 3 distinct literals with equal probability for the literal to be a positive literal or a negative literal.

4.2 Experimental settings and methodology

All the experiments were run on a personal machine with Intel(R) Core(TM) i5-10300H CPU and Ubuntu 20.04.1 LTS. We explore results for the ratio $\frac{L}{N}$ between 3 and 6 with a 0.2 stride for $N = 150$ and we repeat the experiments 100 times and take the median as the result value.

4.3 DPLL implementation performances

We first evaluate the quality of our DPLL implementation on 3-CNF random formulas. In figure 2, we analyze the performances of the DPLL code using the Two Clauses heuristic. We observe that the DPLL solver has difficulties solving formulas a bit before the **satisfiability threshold** that we have noticed for the satisfiability of random formulas at around $\frac{L}{N} \approx 4.5$. Intuitively, this is normal because at that point, formulas are not trivially satisfiable (only a few constraints) or unsatisfiabile (too many constraints)

so the SAT solver takes a lot of time to decide. On the contrary, when a solution has only a few constraint ($\frac{L}{N} < 4$), the SAT solver takes less than a second or a dozen of DPLL calls to find a satisfiable truth assignment, and when ($\frac{L}{N} > 5.2$) the SAT solver takes again less than a second or a dozen of DPLL calls to find a contradiction in the clauses. In the hard interval $4 < \frac{L}{N} < 5.2$, the SAT solver can takes more than 5 seconds and 25.000 DPLL calls to decide the satisfiability problem.

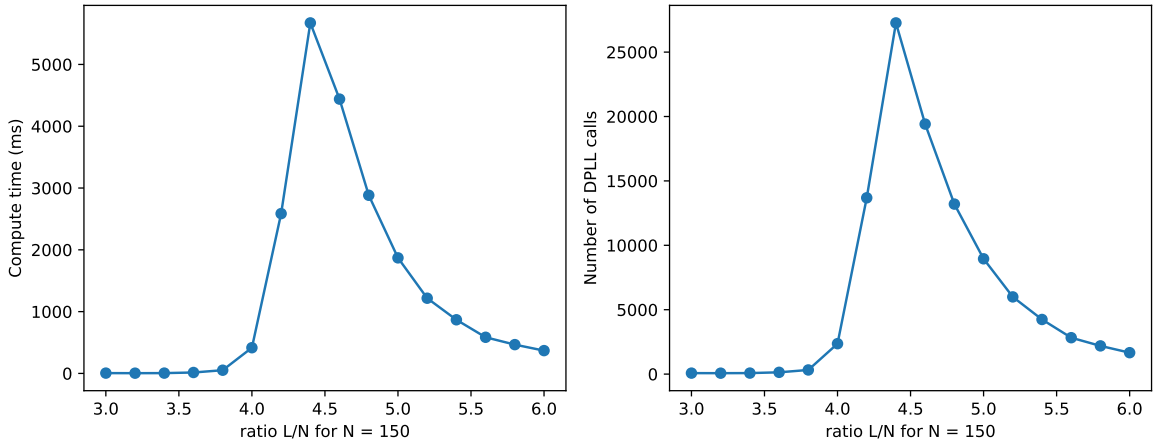

Figure 2: Compute time and number of DPLL calls on random formulas as a function of L/N with N = 150 (Two Clauses heuristic).

4.4 Comparison between heuristics

We now compare 3 distinct heuristics for the Splitting Method :

- The Random Choice heuristic randomly chooses a literal in the *unassigned* list of literals.
- The Two Clauses heuristic chooses the unassigned literal that occurs the most in clauses made of two unassigned literals and break ties randomly. If no such literal exists, it defaults to the random choice heuristic.
- The Jeroslow-Wang heuristic [4] chooses the literal that maximizes the sum $J(x) = \sum_{x \in c, c \in \varphi} 2^{-|c|}$ where φ is a 3-CNF formula and we iterate over the clauses c of φ.
  The idea behind this heuristic is to select literals that occur in shorter clause and to be easy to compute. I have chosen this heuristic to be my heuristic to compare against the 2 previous heuristics.

4.4.1 Jeroslow-Wang vs Random

On figure 3, we observe as expected that both in term of time and number of DPLL calls, the Jeroslow-Wang heuristic severely outperforms the Random Choice heuristic. The

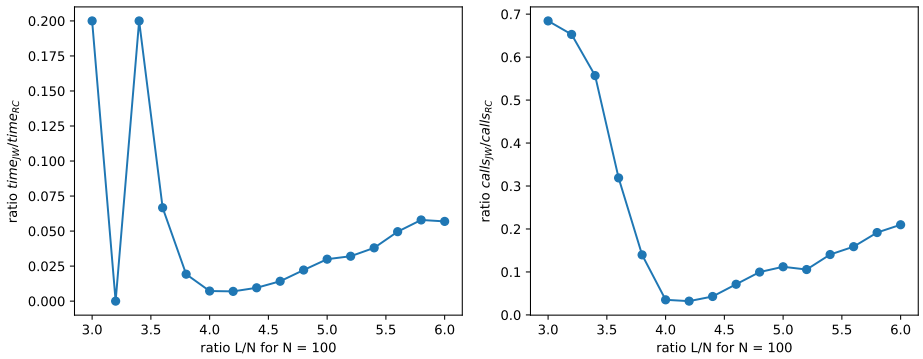

Figure 3: Compute time (left graph) and number of DPLL calls (right graph) ratio between the Jeroslow-Wang (JW) and the Random Choice (RC) heuristics on random formulas as a function of L/N for N = 100

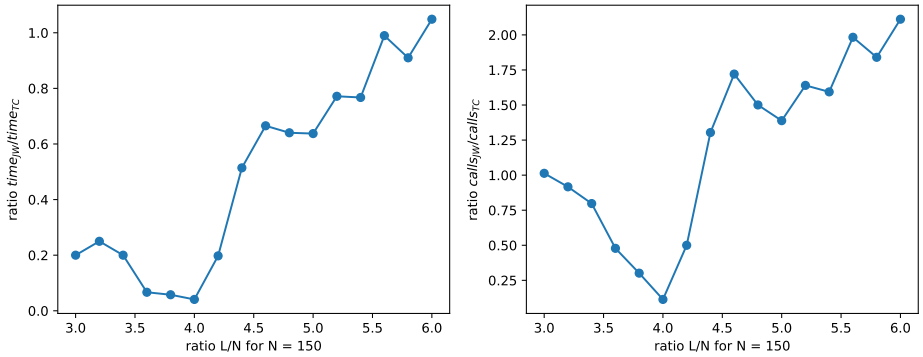

Figure 4: Compute time (left graph) and number of DPLL calls (right graph) ratio between the Jeroslow-Wang (JW) and the Two Clauses (TC) heuristics on random formulas as a function of L/N for N = 150

choice of only 100 variables is justified by the fact that the Random Choice performs so badly that it would take too long to run the experiments on it with 150 variables. In the difficult interval for $\frac{L}{N}$ between 4.2 and 5.2, the Random Choice needs more than 10× more DPLL calls than the Jeroslow-Wang heuristic. As for the time difference, in the difficult interval the Random Choice takes 14× more time than the Jeroslow-Wang heuristic. This was to be expected given that the Random Choice is one of the worst heuristics possible, not smart at all, whereas the Jeroslow-Wang provides some intuition by choosing the literal that occurs in the shortest clauses.

4.4.2 Jeroslow-Wang vs Two Clauses

On figure 4, we notice that the Jeroslow-Wang heuristic always performs better than the Two Clauses heuristic on the time performance. It is important to notice that for the hardest interval (between 4 and 5.2), the Jeroslow-Wang heuristic beats the Two

Clauses heuristic by more than 20%, which is non-negligible. However, we we come closer to $\frac{L}{N} = 6$, their performances are pretty similar. This is because there are more two clauses at that point and the two clauses heuristic is therefore more efficient. On the other hand, the call graph (on the right) shows that for a ratio $\frac{L}{N}$ greater than 4.2, the Jeroslow-Wang heuristic makes more DPLL calls than the Two Clauses heuristics. This is because in that case taking the variables in the two clauses is more effective than what the Jeroslow-Wang heuristic does, i.e. looking for the literal that appears in the smallest clauses. However, computation time of the Two Clauses heuristic is significantly higher than the Jeroslow-Wang heuristic, and that is why this overhead in number of DPLL calls for Jeroslow-Wang is not translated into a time overhead in the time graph.

An interesting question that needs to be raised is whether or not the Jeroslow-Wang heuristic is easier to compute than the Two Clauses heuristic. This could be my implementation choices that favor the Jeroslow-Wang heuristic. However, in that case, I do not believe that this is the case as both heuristics take advantage in similar ways of the implementation design.

5 Conclusion

In this report, I have explained the design choices behind my iterative implementation of a DPLL SAT solver. I have verified that my SAT solver was correct by solving the Einstein's puzzle with it. I have plotted the satisfiability graph for random formulas that points out an inflection point at around $\frac{L}{N} = 4.2$ and models how difficult it is to solve satisfiability on 3-CNF random formulas in function of $\frac{L}{N}$. Finally, I have presented a new heuristic, the Jeroslow-Wang heuristic, that outperforms both the Random Choice and Two Clauses heuristics for all $\frac{L}{N}$ ratios between 3 and 6. Even though the Jeroslow-Wang needs more DPLL calls than the Two Clauses heuristic, it is simple enough to be computed more quickly than the Two Clauses heuristics and that is why it outperforms the Two Clauses heuristic.

6 Takeaways from the project

Even though this project has required a lot of work, mainly because I had to re-implement a SAT solver once I found out that my recursive implementation was not perfoming well, I have enjoyed working on this project. I have been able to practice my skills at transforming a recursive algorithm into an iterative algorithm using a stack. I have also developed my skills on backtracking algorithms and I believe this is a major takeaway as I will surely have to implement this design pattern in the future for my research projects. I am satisfied with the state of my code, that is well-structured and I have been able to know more about the inside of a SAT solver. This is particularly

interesting to me as I have been using their cousins, SMT solvers, a lot for verifying algorithms lately with proof assistants like TLAPS. I am certain that I will take advantage of knowing how a SAT solver works for my future projects.

References

[1] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[2] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[3] Olivier Dubois, Yacine Boufkhad, and Jacques Mandler. Typical random 3-sat formulae and the satisfiability threshold. *arXiv preprint cs/0211036*, 2002.

[4] Robert G Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4):167–187, 1990.