CS4244 SAT Solver Report

Tiong YaoCong, Huang Lie Jun

A0139922Y, A0123994W

yaocong@u.nus.edu, hliejun@u.nus.edu

This document describes the analysis of our SAT Solver design, approaches, results analysis and our reflection and learning points from this project.

1. Setup and Installation

We have written bash script, `run.sh`, in order to aid the execution of our project program and run the different approaches we have designed. `run.sh` is available under `src` folder. The script will be able to compile, run and save the previous run result into the folder `previous-result`.

The following are the instructions on how to use the script :

- `./run.sh` will run the default setup for our project which will be using `AllClauseSolver` to solve our formulated einstein puzzle.
- `./run.sh BENCHMARK` will initialize the project to run in benchmark mode, which will then run all the cnf files in the benchmark folder, test/testcases/benchmark.
- `./run.sh <STRATEGY> <cnf file/folder>` will run the given strategy and cnf file or folder.
    - Available strategies :
        - `DPLL`
        - `RDPLL`
        - `RESOLUTION_CDCL`
        - `TWOCLAUSES_CDCL`
        - `ALLCLAUSES_CDCL`
        - `CHAFF_CDCL`

    Examples :
    - To run with cnf folder : `./run.sh DPLL ../test/testcases/benchmark/50V_218C_sat`
    - To run with cnf file : `./run.sh DPLL ../test/testcases/benchmark/50V_218C_sat/1.cnf`

2. Code Design

Our SAT Solver is written in Java JDK 8 with an Object-Oriented (OO) design.

2.1. Data Structures

2.1.1. Literal, Clause and Clauses

After parsing the CNF file, we represent the formula using a `Clauses` object. `Clauses` is a set of `Clause` that tracks literal and symbol counts when created. It provides convenience methods to inspect the state of the formula, such as obtaining a frequency table of its constituent literals, testing the formula for conflicts given a table of assignments or evaluating the truth value of the formula. The truth value is obtained through the logical conjunction of all of its constituent clauses. A `Clause` is then a set of literals, and its truth value is the logical inclusive disjunction of its constituent literals. A `Literal` is an object that represents a logical symbol and the sign of its instance.

Notably, `Clauses` and `Clause` objects are comparable by size through implementing the `Comparable` generics interface. This allows us to prioritise or order clauses to process smaller clauses firsts. On another note, `Clause` and `Literal` are made hashable through overriding the `Object` hashcode. This allows us to use them as index or keys in hashed tables for effective lookup.

*Note that all timings include time writing output to file.

2.1.2. Implication Graph

The `IGraph` object contains a variable graph that represents literal assignment (variables) as nodes and antecedent clauses as edges. The `IGraph` hence models an implication graph and maintains the state of the decisions and implications made in Conflict-Driven Decision Learning (CDCL) solvers. It also offers convenience methods to add branching or implied variables, as well as to revert to a certain decision level state when backtracking. The CDCL approach will be evaluated later in this report.

2.2. Solvers

In this project, we are expected to make quantitative assessments on the efficiencies between different SAT solvers, specifically DPLL and CDCL, as well as between the different heuristics for strategies used in modern CDCL solvers. Instead of writing the solvers as separate classes, we utilised subclass substitutivity and polymorphism to invoke common solver methods for assessment and timing.

Moreover, we used inheritance and method overriding to split the key implementation differences across different solver subclasses. Besides saving effort on rewriting common methods and benchmarking, the subclassed solvers aptly accentuates the differences between each solving technique which makes it very helpful for qualitative comparison and analysis.

3. Solver Variants and Techniques

Our approach is to implement a few heuristics and compared the design and the execution time for each implementation.

3.1. DPLL

We initially started with the implementation of `Davis-Putnam-Logemann-Loveland (DPLL)` algorithm as it is a complete, backtracking-based search algorithm which simplifying the formula and recursively checking if the formula is satisfiable. Hence, this approach will allow us to ensure that our initial algorithm and understanding of the problem is on the right track.

```
function DPLL(Φ)
   if Φ is a consistent set of literals
       then return true;
   if Φ contains an empty clause
       then return false;
   for every unit clause {l} in Φ
      Φ ← unit-propagate(l, Φ);
   for every literal l that occurs pure in Φ
      Φ ← pure-literal-assign(l, Φ);
   l ← choose-literal(Φ);
   return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {not(l)});
```

By following the algorithm,we have designed two design: Recursive DPLL and Iterative DPLL. Both design have similar approach where it uses the `Assignment` class to keep track of the current assigned variables in order to determine if the variables have been assigned before and to decide to flip the value of the variable.

In the above pseudocode, unit-propagate(l, Φ) and pure-literal-assign(l, Φ) are functions that return the result of applying unit propagation and the pure literal rule, respectively, to the literal l and the formula Φ. In other words, they replace every occurrence of l with "true" and every occurrence of not l with "false" in the formula Φ, and simplify the resulting formula. The or in the return statement is a short-circuiting operator. Φ ∧ {l} denotes the simplified result of substituting "true" for l in Φ. Thus, allowing the DPLL function to return when a partial satisfying assignment is found. Hence, we have implemented a naive heuristic: `heaviest weighted variable` for both DPLL implementation. Every variable are weighted with a weight based on the number of occurrences in the given CNF form. The design will utilize the `heaviest weighted` heuristic to choose the next heaviest unassigned variable for the solver to solve.

3.1.1. Recursive DPLL

Recursive DPLL's strategy is to clone `assignment` class at every stages of evaluation. Hence, when a conflict variable is found, the recursive call will end and backtrack to the state where the recursive function is invoked. This will

*Note that all timings include time writing output to file.

allow the algorithm to reset the variable and continue to try out different combination of values.

Our implementation also allows partial evaluation which has been described under section `3.1`.

3.1.2. Iterative DPLL

Iterative DPLL has an identical approach as recursive DPLL. However, insufficient memory problem may arise where limited memory is in place.

Hence, we have re-designed our recursive DPLL solver into an iterative-DPLL solver in order to increase the efficiency of the solver..

3.2. CDCL

DPLL solvers are exceptionally slow considering that it operates on a chronological backtrack schema. In CDCL, conflict-driven clause learning is introduced. This approach allows for non-chronological backtracking triggered by conflicts. The following is a sketch pseudocode that describes the CDCL algorithm:

Algorithm 1 Typical CDCL algorithm

```
CDCL(φ, ν)
 1  if (UnitPropagation(φ, ν) == CONFLICT)
 2     then return UNSAT
 3  dl ← 0                                   ▷ Decision level
 4  while (not AllVariablesAssigned(φ, ν))
 5        do (x, v) = PickBranchingVariable(φ, ν)          ▷ Decide stage
 6           dl ← dl + 1                   ▷ Increment decision level due to new decision
 7           ν ← ν ∪ {(x, v)}
 8           if (UnitPropagation(φ, ν) == CONFLICT)        ▷ Deduce stage
 9              then β − ConflictAnalysis(φ, ν)            ▷ Diagnose stage
10                   if (β < 0)
11                      then return UNSAT
12                      else Backtrack(φ, ν, β)
13                           dl ← β          ▷ Decrement decision level due to backtracking
14  return SAT
```

While there are numerous variants of CDCL solvers and a spectrum of modern CDCL strategies and techniques, we chose to focus on the following 3:

3.2.1. Variable Branching Heuristics

Picking decision variables is a critical phase in CDCL solvers. It determines how many or little implications can be made due to the frequency in occurrences of the logical symbol and how quickly a conflict can be arrived at. The heuristics that we have tested are the following:

Random: Pick an unassigned variable randomly from a list of unassigned symbols. This approach is used in `ChaffSolver`.

Most frequent across 2-clauses: Pick an unassigned variable that occurred the most across 2-clauses. This heuristic forces 2-clauses to become assertive after decision-making. This approach is used in `TwoClauseSolver`.

Most frequent across all clauses: Pick an unassigned variable that occurred the most across all clauses. This approach is used in `AllClauseSolver` and `ResolutionSolver`.

3.2.2. Decision and Unit Propagation

As the CDCL solver picks decision variables, the decision is used to propagate assertive clauses for implied variables. For each decision that is made, a decision node (vertex) is created in the implication graph. For every variable that it implies, a node is created for the implied variable with a directed edge connecting from the decision node to the implied node. The edge captures the antecedent clause that is responsible for asserting the implication.

3.2.3. Conflict Analysis

When a conflict is encountered while propagating a decision, we need to identify the cause of the conflict and learn never to land ourselves in the same situation. In our approach, we adopted Chaff's first Unit Implication Point (1-UIP) schema. Specifically, we used the implication graph to extract antecedent clauses to apply resolution with, starting from the conflict clause antecedent to the conflict node. However, the resolution is terminated as soon as we have reached a state where only 1 variable in the resolved clause belongs to the conflict level. This provides us with an assertive clause that can be propagated after backtracking to the highest level of the remaining variables in the resolved clause.

*Note that all timings include time writing output to file.

4. Solvers Results and Analysis

4.1. Timing and Performance

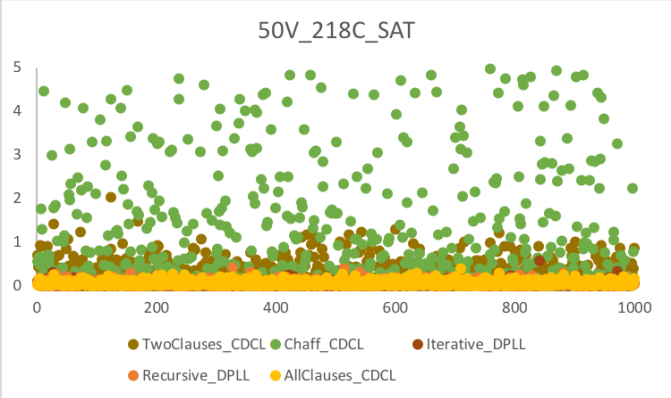

Figure 1 : 50V_218C_SAT Comparison

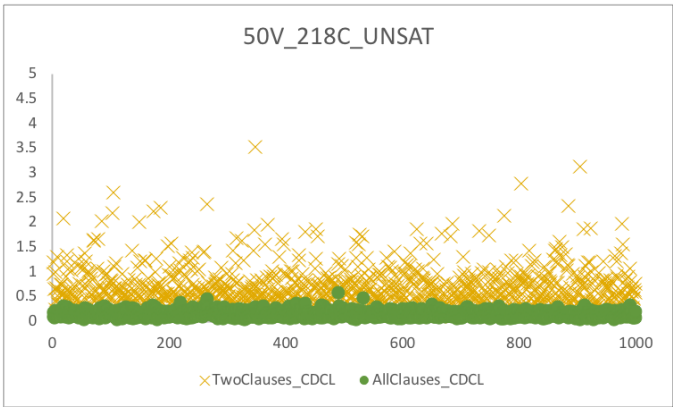

Figure 2 : 50V_218C_UNSAT Comparison

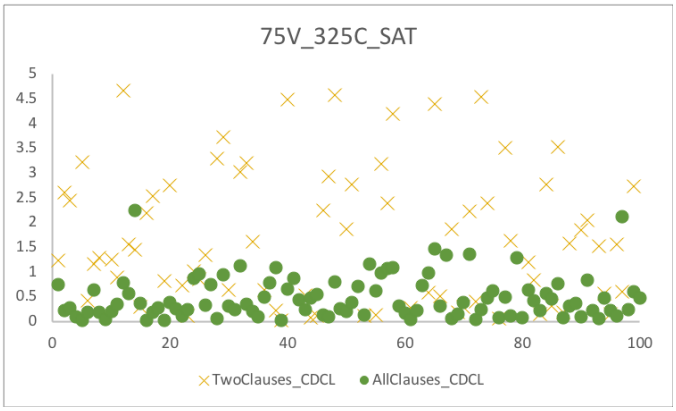

Figure 3 : 75V_325C_SAT Comparison

We have run our solvers through a series of benchmarking test and convert our result into a scatter plot graph for easier comparison. From the above figure 1, we can observe only `ChaffClauseSolver` which has random results across the benchmarking test. The result from `ChaffClauseSolver` is expected due to its randomness implementation.

While running through the benchmarking test, we have observed that both implementation of `DPLL` have shown significantly slower result as compared to all implementation of `CDCL` solvers. Hence, we decided to scrap it away from our benchmarking test for better analysis and comparison.

We can also observed that our current best solver is `AllClauseSolver` which is both stable and most efficient among all the other solvers.

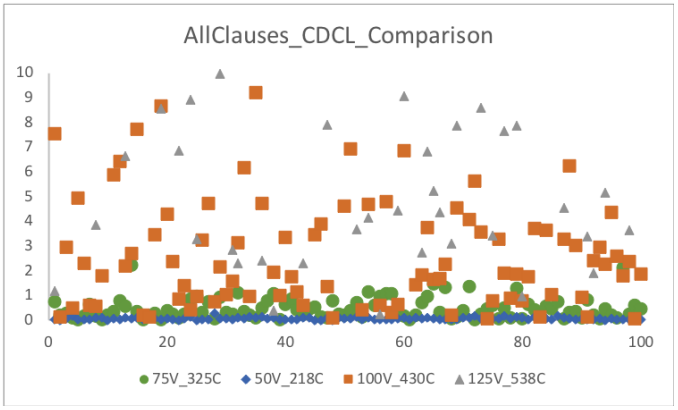

Figure 4 : AllClauses_CDCL Comparison

In figure 4, we compared the different results from `AllClauseSolver`. We can observe that it is indeed stable as the results from each series are closely related to each other. Moreover, we can see that the execution time increases with the number of variables and clauses.

4.2. Variations & Problems Faced

We have implemented five different types of solvers and have tested the solver rigorously against `SATLIB`[1] benchmark problems. For more result logs, you may refer to `/result` to view the comparison of different implementation against the same cnf file with different number of variables and clauses.

During the implementation and testing, we came out with different heuristics such as sorting based on clauses similarity and size of clauses that we initially thought that they were good and our solver

*Note that all timings include time writing output to file.

will be able to outperform our expectations. However, we are mistaken by our naive thoughts. The results were in the range of 7000+ seconds for test cases in the 100 variables 430 clauses.

Upon researching, we came upon a few heuristics in different research papers. We implemented different heuristics and went through several testing phases and came out with our current best solver, AllClauseSolver. With the new heuristic implementation, the result improved drastically by over 1000%.

From the figures in section 4.1, we can observe all the different implementations of our solver. However, we can also observe that ALLClauses_CDCL solver is not just the most efficient solver but is also stable and the execution time grow linearly as the number of variables and clauses increase which can be observed in the figure 4 under section 4.1.

5. Einstein Puzzle Encoding

To model the Einstein Puzzle, we have manually translated the puzzle into a series of clauses in cnf form. The puzzle in their variable name can be found at ./puzzle/einstein_strict.cnf. The integer form can be found at ./puzzle/einstein.cnf. The puzzle is modelled with the assumption that the green house has to be strictly on the immediate left of the white house. An alternative model for non-strict relation between the green and white house can be found at ./puzzle/einstein_new.cnf. Note that einstein.cnf is translated from einstein_strict.cnf by replacing P with 1, Q with 2, R with 3, S with 4 and T with 5.

5.1. Variables and Design

To model the puzzle, we declare 150 variables to capture all possible states. Our initial approach of using 30 variables was unsuccessful as it only captures a single person to be true, which leaves the remaining 4 persons to be vacuously true and hence potentially violating the rules set in place for the puzzle. Suppose that we have each house owner tagged to an ID, then for each person, x in {P, Q, R, S, T}, x can take values x1 to x30.

The following table maps the variables:

ID	House	Color	Nationality	Drink	Cigar	Pet
P	1: 1st	6: red	11: Brit	16: tea	21: Pall Mall	26: dogs
Q	2: 2nd	7: green	12: Swede	17: coffee	22: Dunhill	27: birds
R	3: 3rd	8: yellow	13: Dane	18: milk	23: Blends	28: cats
S	4: 4th	9: blue	14: Norwegian	19: beer	24: Bluemaster	29: horse
T	5: 5th	10: white	15: German	20: water	25: Prince	30: fish

5.2. Rules Translation

The rules defined in Einstein Puzzle was modelled into 1300 clauses, with the following distribution:

simple rule clauses = 100
proximity rule clauses = 570
exactly one rule clauses = 330
unique rule clauses = 300

Single rule clauses refer to the following 10 rules:

The Brit lives in the red house.
The Swede keeps dogs as pets.
The Dane drinks tea.
The green house's owner drinks coffee.
The person who smokes Pall Mall rears birds.
The owner of the yellow house smokes Dunhill.

*Note that all timings include time writing output to file.

The man living in the center house drinks milk.
The Norwegian lives in the first house.
The owner who smokes Bluemasters drinks beer.
The German smokes Prince.

Suppose x is a property in set {color, nationality, drink, cigar, pet} and suppose y is also in that set but x != y. Then, proximity rules are defined by specifying the constraints between person P and person Q, where P and Q are different persons, such that P and Q have constraints in the houses that they own when person P has a property x and person Q has a property y. Proximity rules refer to the following 5 rules:

The green house is on the left of the white house.
The man who smokes Blends lives next to the one who keeps cats.
The man who keeps the horse lives next to the man who smokes Dunhill.
The Norwegian lives next to the blue house.
The man who smokes Blends has a neighbor who drinks water.

Exactly-1 rules define exclusive disjunction relations within each category. For example, person P can only live in exactly one house:

$$P1 \oplus P2 \oplus P3 \oplus P4 \oplus P5 = 1$$

Unique rule clauses define the unicity of property or traits ownership. For instance, we know that only 1 person can live in the first house. If P1 is the owner of the first house, the following stands:

$$P1 \rightarrow \neg (Q \rightarrow 1)$$
$$P1 \rightarrow \neg (R \rightarrow 1)$$
$$P1 \rightarrow \neg (S \rightarrow 1)$$
$$P1 \rightarrow \neg (T \rightarrow 1)$$

5.3. Performance

The Einstein Puzzle took an average of 3 seconds to solve. The following is the result:

[11=true, 12=false, 13=false, 14=false, 15=false, 16=false, 17=false, 18=true, 19=false, 21=false, 22=false, 23=false, 24=false, 25=true, 26=false, 27=false, 28=false, 29=false, 31=false, 32=true, 33=false, 34=false, 35=false, 36=false, 37=false, 38=false, 39=true, 41=false, 42=false, 43=false, 44=true, 45=false, 46=false, 47=true, 48=false, 49=false, 51=false, 52=false, 53=true, 54=false, 55=false, 56=true, 57=false, 58=false, 59=false, 110=false, 111=false, 112=false, 113=false, 114=true, 115=false, 116=false, 117=false, 118=false, 119=false, 120=true, 121=false, 122=true, 123=false, 124=false, 125=false, 126=false, 127=false, 128=true, 129=false, 130=false, 210=true, 211=false, 212=true, 213=false, 214=false, 215=false, 216=false, 217=false, 218=false, 219=true, 220=false, 221=false, 222=false, 223=false, 224=true, 225=false, 226=true, 227=false, 228=false, 229=false, 230=false, 310=false, 311=false, 312=false, 313=true, 314=false, 315=false, 316=true, 317=false, 318=false, 319=false, 320=false, 321=false, 322=false, 323=true, 324=false, 325=false, 326=false, 327=false, 328=false, 329=true, 330=false, 410=false, 411=false, 412=false, 413=false, 414=false, 415=true, 416=false, 417=true, 418=false, 419=false, 420=false, 421=false, 422=false, 423=false, 424=false, 425=true, 426=false, 427=false, 428=false, 429=false, 430=true, 510=false, 511=true, 512=false, 513=false, 514=false, 515=false, 516=false, 517=false, 518=true, 519=false, 520=false, 521=true, 522=false, 523=false, 524=false, 525=false, 526=false, 527=true, 528=false, 529=false, 530=false]

From this result, it shows that the person S, who is a German (415 = true), owns the fish (430 = true).

6. Refutation Proof

To certify UNSAT, we need to prove that the formula is contradictory. In our approach, we used the CDCL Solver to generate resolution proof, taking advantage of the learnt clauses resolution and propagation.

*Note that all timings include time writing output to file.

6.1. Learnt Clauses

In CDCL, we derive learnt clauses from conflict clause resolution with antecedent clauses. `ResolutionSolver` does the same, but this time round, it captures the set of antecedent clauses that were used in resolution.

6.2. Resolution Rule

After we derived a learnt clause, we backtrack to a suitable level and propagate the learnt clause. However, should a conflict occur, it would reflect that the clauses learnt are conflicting. We thus make use of the set of antecedent clauses that are used for resolution to find a sequence of resolution that would lead to an empty clause that alludes to contradiction. This is done so by applying resolution rule to every possible pair of clauses and adding the resolved clause to the set for future matches.

6.3. Resolution Tracing

While we apply the resolution rule to every possible pair, we record the workings of the resolution by mapping the resolvents to the resolved clause. This allows us to recall and back trace the resolution once we have obtained the empty clause.

6.4. Thresholds

To speed up the resolution, we imposed a hard threshold of max. 5-clauses. That is, we ignore every resolved clause that have more than 5 literals. This keeps the active resolvent set from exploding too quickly and discards the larger clauses that are less likely to contribute to deriving an empty clause. We find that a better alternative would be to use a dynamic threshold that responds to the number of variables and number of clauses in the formula.

6.5. Performance

The performance of the refutation proof varies largely due to the complexity of the refutation. A possible improvement that we can make is to record the resolution workings while learning the clauses, instead of only the antecedent clauses. The following are 2 examples of running the refutation proof on UNSAT cnf formulas with 50 variables and 218 clauses:

test/testcases/benchmark/50V_218C_unsat/1.cnf: 4.603 seconds

v 3
-1 ()
1 (14 v 19 v -46)
2 (-14 v -19 v 46)
1 2 -1

test/testcases/benchmark/50V_218C_unsat/3.cnf: 2.082 seconds

v 3
-1 ()
1 (14 v -34 v -36)
2 (-14 v 34 v 36)
1 2 -1

*Note that all timings include time writing output to file.

7. Reflection

7.1. Tiong YaoCong

My analysis of the project would be that the CDCL offers a much more chances of optimizing it based on different type of heuristics which are the core of optimizing the solver while DPLL is just a naive solver which could be use to proof correct result instead.

My greatest takeaway from this project and module would be that to not to be complacent and to explore and understand further when in doubt. I remembered that during the lesson, someone mentioned that this module is not teaching anything, we are not learning anything. Initially, I thought that I have the same thought as them. However, after going through this project, my mindset totally changed. I would have to disagree with them. I have learnt and experience something interesting that I would never have experienced it without this project.

Initially, I thought that we would have implement something that is being taught in all the prerequisite classes since Year 1. However, as all our naive thoughts get crashed by the raw result. I realised that I do not have a strong foundation in this and this project allows me to explore many new interesting heuristics. I managed to discover several interesting heuristics that I would have never thought about. I also came to realise that heuristic is the core of the project, without a properly designed heuristic, the outcome of the system will never be ideal.

I understand your frustration in teaching this module where most of the students are not giving much effort and thoughts. Hence, I sincerely apologise and thank you for this great opportunity to actually "learn" rather than having going through some new content that we may never explore it ever again or forgotten in a few years time.

I would assign myself a grade B+ for my work, there are much more to be done to refine the system.

7.2. Huang Lie Jun

My intuitive analysis of the project would be that the CDCL offers much more optimisation opportunities with its conflict-driven paradigm. With the branching and conflict analysis heuristics, we are merely scratching the surface of the CDCL solver. While there exists many heuristics for branching, such as Exponential Recency Weighted Average (ERWA) and VSIDS (Variable State Independent Decaying Sum) that we have yet to explore, our result with the general success of most frequent all-clauses heuristics is indicative towards the need to target the most frequently occuring literal, and it would be more effective should we consider the recency apart from just the frequency of the literal.

My greatest takeaway from this project would be to have a holistic understanding of the functioning of SAT solvers, from the different strategies of DPLL and CDCL, to the heuristics used to optimise the performance of CDCL, as well as the use of these solvers in solving problems that are modelled in CNF form. The most excruciating was to compose the 1300 clauses from scratch for the Einstein Puzzle, but that in itself says alot about the complexity of modelling real-world problems in propositional logic, and the dire need to optimise and increase the efficiency in these constraints encodings.

I would assign myself a grade B+ for my work, as it could use several improvements, such as search restarts and clause freezing or removal, which we ran short of time for.

*Note that all timings include time writing output to file.

8. References

[1] SATLIB - Benchmark Problems. (n.d.). Retrieved from http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

[2] Liang J., Ganesh V., Poupart P., & Czarnecki K. (n.d.). *Exponential Recency Weighted Average Branching Heuristic for SAT Solvers* [PDF].

[3] Audemard, Gilles, et al. "On Freezing and Reactivating Learnt Clauses."

[4] Tichy,, R., & Glase, T. (2006, April 25). *Clause Learning in SAT Seminar Automatic Problem Solving*[PDF].

[5] Marques-Silva, Joao, et al. "Conflict-Driven Clause Learning SAT Solvers." 2008.

[6] *Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers* [PDF]. (2015, September 14).

[7] Sinz,, C., & Balyo, T. (2016, June 27). *Practical SAT Solving*.

*Note that all timings include time writing output to file.