CNF SAT Solver

CS4244 Project
Mukesh Gadupudi (A0161426L)
Tiyyagura Hyma (A0158082A)

Introduction

This paper describes the implementation of a SAT Solver using the Conflict Driven Clause Learning (CDCL) algorithm to check the satisfiability of an input Conjuctive Normal Form (CNF) formula in DIMACS format. The SAT Solver returns a satisfying assignment, and UNSAT otherwise if there is no satisfying assignment for the input CNF formula.

Setting Up and Running our SAT Solver

To run our SAT Solver, download the program and open it using IntelliJ. Then, under configurations for 'Run', set it to CDCL (CDCL.java). This is our main class that runs the SAT Solver. Once running, you would have to input, into the run terminal, the file path of the cnf file containing the formula. We have included some cnf files in our src/cnf_test folder. Our SAT Solver would then output SAT or UNSAT, for satisfiable and unsatisfiable input formulas respectively. For UNSAT formulas, it would also output the proof of unsatisfiability into the file resolutionProof.txt. More details about this will be in a later section under Project 2.

CDCL Algorithm

The very essence of a CDCL algorithm, as the name suggests, is to speed up the solving process by learning clauses from conflicts and in turn, enabling backtracking.

The difference between a DPLL solver and a CDCL solver is the introduction of something called non-chronological backtracking or backjumping. The idea behind it is that often, a conflict is caused by a variable assignment that happened much sooner than it was detected, and if we could somehow identify when the conflict was caused, we could backtrack several steps at once, without running into the same conflict multiple times.

This paper describes the implementation of the CDCL algorithm and various heuristics over it, which further increase the speed of the solver.

The algorithm is as follows:

- Perform unit propagation, in cases wherever possible, to assign values to some of the variables.
- We start at decision level zero and increment the decision level every time that we guess and set a variable value.
- Thereafter, if there exists at least one variable without an assignment yet, we pick a branching variable, according to one of the branching variable heuristics that will be detailed below, and assign it a value of either True or False.
- Unit propagation is performed again, every time after we assign or set a variable.
- If a conflict is reached, we would backtrack again to the desired backtracking level. This back tracking level is decided based on an analysis of the conflict which will be called CONFLICT ANALYSIS. A conflict is reached when unit propagation forces the value of a previously assigned variable to be different from the value that it was assigned.
- After backtracking, we iteratively continue this process and if at any point the back tracking level is less than zero, we return UNSAT immediately as this signifies that there is no possible satisfying assignment for the input formula.

The basic pseudo code of the CDCL algorithm is shown [1]:

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
```

Term Definitions

Unit Propagation

Unit propagation occurs if it follows that a certain variable can ONLY take 1 value. In the case of 1-literal clauses, it is apparent that the clause would have to evaluate to true for the formula to be satisfied, and therefore the single literal in that clause has to evaluate to true as well. In such a case, we assign the value of it to be true. After some variables have

already been assigned, if there is a clause with 1 unassigned variable and the other variables assigned have been assigned false, then it also follows that that unassigned variable has to be assigned true, to satisfy the clause and in turn, the formula. Unit propagation speeds up the SAT Solver greatly as it allows values to propagate quickly and conflicts are also reached quickly, since in many cases, unit propagation causes 2 different clauses to assign the same variable to different values. In this case, a conflict is reached.

Resolution

Resolution is a technique used to speed up the solving of SAT Solvers as new clauses can be learnt. A single clause is learnt from 2 clauses which contain complementary literals, meaning that one is a propositional variable and the other is a negated propositional variable. For example, x_1 and $\neg x_1$ are complementary literals. The resulting clause that is learnt contains all the literals in the 2 clauses, except for the complementary literals which are 'removed'. The remaining literals are joined by $\vee$ as per normal.

BackTracking

Backtracking is a procedure in which after a conflict is found, we change the value of a guessed variable to its opposite value to try that value out. The value it backtracks to depends on the scheme, which will be elaborated in a later section.

Data Structures

Parsing

Variable Variable refers to a literal object. It contains two parameters Name and Value. The name can be positive or negative integers representing a positive literal or its negated representation respectively. Value can take either True or False value depending on the situation. These parameters have their Getters, Setters and Comparatives defined. We have also added helper functions such as modVariableName() for cases where we need to compare the values of a negated literal and the actual literal.

Clause Clause represents a single disjunction of Variables. Each clause has a list of Variable objects called 'OrVariables' which represents the Variables in a disjunction. A clause object has its own Getter, Setter and Helper functions for it to interact with other objects.

Parser Input parser parses the input file given in DIMACS format.It initially reads the file and splits the file based on ending zeros and initial line codes. It breaks down the whole file into a formula which is essentially ArrayList of Clauses.

Heuristics

There are several ways of implementing the above mentioned CDCL algorithm. We discuss in detail about the different methods of choosing a branching variable, conflict analysis and backtracking.

Branching Variables

Picking a branching variable and assigning a truth value to it proves to be of great importance as it can speed up the solving time if a good heuristic is implemented.The number of possible heuristics is virtually unlimited, especially if we count various hybrids and small tweaks, but in this report, we will look discuss at 2 major ones.

Random Branching The baseline model that we first implemented was the Random Branching Variable (Baseline Model).
All choices (unassigned variables) are resolved randomly with random distribution and the truth value assigned to it would be random as well.

2 Clause Heuristic Propositions with maximum occurrences in 2 literal clauses are chosen and ties are broken randomly. Such a heuristic forces the two clauses to become assertive and hence would speed up the solver greatly. Fixing the value of a variable in a 2-literal clause would allow unit propagation to occur if the value of that variable fixed is false, forcing the only other variable in the clause to be set to true. This would allow the value of many other variables would be set, speeding up the solving process.

All Clause Heuristic We chose the most frequently occurring unassigned variable to fix its value to false. This would speed up the solving process as fixing the value for the variable that occurs the most might cause the unit propagation of other values, or fixing the value of that particular variable might satisfy the clause and hence be able to find a satisfying assignment to the entire formula faster.

Conflict Analysis

Analysing a conflict and deciding the level to backtrack proves to be a very important part of CDCL algorithm. This part also enables learning the correct kind of clauses which force variables to take values much faster than expected. This helps in speeding up the process of identifying the satisfiablity of a formula.In this report we will be looking at two major implementations of Conflict Analysis.

GRASP In GRASP Clause Learning when the first conflict is discovered, the decision level would remain the same and the value of the variable that is guessed at that decision level would be reversed. Backtracking to the previous decision level would only occur when the reversed variable value also yields a conflict. However, the drawbacks of GRASP is that it is costly in terms of time and space as it depends on the total number of variables and learns many new clauses.In the GRASP backtracking scheme, clauses are learnt at all the Unit Implication Points (UIPs) and a global clause, to ensure that at the current decision level, both branches are made with respect to the same variable.

Chaff-UIP In Chaff-UIP heuristic implementation, the decision level to back track would be decided based on the final clause learnt.

The initial conflict clause is resolved with the antecedent of the last propogated variable in the conflict clause. This is done iteratively until a terminating condition is reached. The clause is identified as a terminating or final clause if it contains only one variable from the current decision level of conflict.

Once the final clause has been learnt the it backtracks to the second highest decision level among the variables in the final clause. This makes the clause an asserting clause and hence makes the variable flip its value.

Implementation

Once the path of the file is given it is input into the Parser to identify the file and parse it into a Formula. It identifies the total number of variables and total number of clauses from the file.

Once the formula has been created, it is passed into the file CDCLSolverUpdated.java which is where the actual algorithm behind CDCL has been implemented.

The heuristics have been implemented in such a way that they can be set as a parameter so as to evaluate their performance. To toggle between the different heuristics, you can manually change the heuristic in CDCL.java, in the following line:
String solution = cdclSolver.solution(”Random”, ”GRASP”, true);

The default options are ”Random” and ”GRASP”, representing random branching variable and GRASP backtracking. The last field will be explained in a later section.
The first parameter which represents the heuristics for branching variable can toggle between

- ”Random”
- ”2C”
- ”AllC”

The second parameter which represents the heuristics for conflict analysis and back tracking can toggle between

- ”Grasp”
- ”1UIP”

Testing

To test the accuracy and speed of our SAT solver, we generated files using these 3 methods:

1. Manually writing simple programs by ourselves, and working the solutions out.
2. Using cnfgen - installing it and running on terminal with the command - cnfgen krandcnf 3 150 100 (where the first field represents number of literals, second field represents number of variables and last field represents number of clauses). Detailed instructions and installation can be found here http://massimolauria.net/cnfgen/ [6]. We also used the php command to generate UNSAT problems and this was especially useful in our debugging as php (pigeonhole principle) problems generally take a long time to run and hence, we would be able to test and see which was the problematic part should it output the wrong result. An example command is cnfgen php 10 7, where the first field represents number of pigeons and second field represents number of holes.
3. Getting source files online (these files can be found in src/cnf_test/uf20-91 and we retrieved these files online from (https://www.cs.ubc.ca/ hoos/SATLIB/benchm.html [7]).

For the smaller and simpler programs, we verified the answers by manually working it out and also using cryptominisat to check for the satisfiability of the formulas. For the bigger problems, we only used the latter method to check for satisfiability as working out the answer manually was too tedious.

Analysis

The results and comparison of the time taken and number of invocations of pickBranchingVariable required for each of the heuristics is shown below. We ran each file 20 times and got the average time taken.

Time Taken

The timings are in milliseconds and the column headers (i.e. (Random, GRASP), (Random, 1UIP), etc.) represent the heuristics implemented. The first field refers to the pickBranchingVariable heuristic (random, 2C or AllC) and the second field refers to the conflict analysis heuristic (GRASP or 1UIP). We analysed the timings for both small and big SAT and UNSAT formulas. The row headers refer to the formula file. simple is a SAT formula with 3 variables while 50Sat is a SAT formula with 50 variables. php1 is an UNSAT formula with 20 variables and 42unsat is an UNSAT formula with 42 variables.

	Random, GRASP	Random, 1UIP	2C, GRASP	2C, 1UIP	AllC, GRASP	AllC, 1UIP
simple	0.6	6	6	4	5	4
50Sat	18	39	86	22	39	21
php1	5.8	13	17	14	6	15
42unsat	156152.5	42926	53478	51024	192983.6	28735

Table 1: Comparison of time taken (ms)

We noticed that when the file size is small, the time taken for our SAT solver actually increased. For example, for a small SAT formula (simple.cnf), which comprises of ... clauses, the time taken for the baseline model was 0.8 ms while the time taken with the All-Clause heuristic and 1UIP heuristic implemented was significantly longer, at 4ms. For a small UNSAT formula (php1.cnf), the time taken for the baseline model was 5.8 ms and this timing increased almost threefold to 15ms for our implemented with the All-Clause and 1UIP heuristics implemented.

This could attributed to the pre-processing overhead required to calculate the frequency of variables in the 2-clause and all-clause heuristic, which would be significant in relation to the number of clauses for the smaller formulas.

However, when the formulas are larger, there was a great reduction in run time. For example, for 42unsat.cnf, which contains 42 variables, the time taken for our SAT Solver (with all-clause heuristic for branching and 1UIP backtracking scheme) reduced by almost 5 times, from 156152.5ms to 28735ms, compared to the baseline model of a random variable branching and GRASP backtracking scheme.

Number of pickBranchingVariable Invocations

	Random, GRASP	Random, 1UIP	2C, GRASP	2C, 1UIP	AllC, GRASP	AllC, 1UIP
simple	2	1	2	2	2	1
50Sat	1832	40	551	51	104	13
php1	37	5	6	7	8	6
42unsat	1735	1973	1722	1702	1730	1528

Table 1: Comparison of number of invocations of pickBranchingVariable

The number of invocations required of pickBranchingVariable decreased significantly for 50Sat, a large SAT formula especially due to the implementation of 1UIP heuristic. When we analyse the performance using 50Sat as the example, we can conclude that 1UIP heuristic is more important of a heuristic compared to the branchingVariable heuristics, as 1UIP caused the number of invocations to decrease significantly. However, we cannot come to a definite conclusion as this finding is not consistent across all the formulas and is specific to 50Sat in this case. For 42unsat, a large UNSAT formula, the number of invocations, though it decreased when AllC and 1UIP heuristics were used, the difference was not that drastic as compared to that for 50Sat and php1.

Summary of Analysis

In summary, the larger the formulas, the more efficient the heuristics would be as the additional pre-processing overhead required for these heuristics would be relatively negligible as the formula gets larger and larger. The number of invocations of pickBranchingVariable decreased as heuristics were implemented, and the impact of the decrease was more pronounced in the bigger formulas. Therefore, for very small formulas, it would be better to stick to the baseline model and only introduce the heuristics for the larger formulas. In the real world scenario, most problems would have an extremely extremely large formula and therefore the heuristics, especially All-Clause branching heuristic and 1UIP conflict analysis heuristic would help to speed up the solver greatly.

Einstein's Puzzle

Our encoding of Einstein's Puzzle can be found at src/PuzzleSolver.java and the corresponding cnf file generated can be found at src/einsteinSolver.cnf.

Encoding

To encode Einstein's Puzzle, we first assigned an integer value to the variables in the puzzle.

To model the nationalities, we assigned each of the 5 nationalities an integer value as follows: brit = 0; swede = 1; dane = 2; norwegian = 3; german = 4.

To model the house colours, we assigned each of the 5 colours an integer value as follows: red = 5; green = 6; white = 7; yellow = 8; blue = 9.

To model the pets, we assigned each of the 4 pets an integer value as follows. The last pet is a fish, which is the question of Einstein's puzzle. dog = 10; bird = 11; cat = 12; horse = 13; fish = 14.

To model the beverage, we assigned each of the 5 beverages an integer value as follows: tea = 15; coffee = 16; milk = 17; beer = 18; water = 19.

To model the type of cigar, we assigned each of the 5 cigar an integer value as follows: pallMall = 20; dunhill = 21; blends = 22; bluemasters = 23; prince = 24.

To model the house positions, we assigned each of the 5 positions an integer value as follows: firstHouse = 105; secondHouse = 106; thirdHouse = 107; fourthHouse = 108; fifthHouse = 109.

Implementation

To encode the uniqueness of each house colour, each nationality, each beverage, cigar and pets, we wrote the following code for the house colour example, and did the same for the rest.

for (int i=0; i¡5; i++)
for (int j=0; j¡5; j++)
print(houseColour(i,j+1) + ” ”);
print(” 0”); → to end the clause
for (int j=0; j¡5; j++)
for (int k=0; k¡j; k++)
print(-(houseColour(i, j+1)) + ” ” + -(houseColour(i, k+1)) + ” 0”);
for (int k=0; k¡5; k++)
if(!(i==k)) → if i!=k, there exists a constraint that both persons cannot live in the same coloured house
print(-(houseColour(i, j+1)) + ” ” + -(houseColour(k, j+1)) + ” 0”);

The above code models the fact that the house colour is unique to the person, and there is a one-to-one matching (i.e. only one person will live in a certain house and a certain house will be inhabited by only one person). If person A lives in the red house, then it follows that: A → ¬(B → redHouse).

Solution

Our SAT Solver returns SAT for Einstein's Puzzle and outputs the following satisfying assignment:

192: true, 120: true, 105: false, 110: false, 115: false, 116: false, 117: false, 118: false, 119: false, 125: false, 60: false, 45: false, 91: false, 92: false, 93: false, 94: false, 95: true, 80: false, 85: false, 90: false, 100: false, 15: false, 35: false, 25: false, 5: false, 30: false, 22: false, 122: false, 153: true, 151: true, 7: false, 52: false, 13: false, 78: false, 11: false, 76: false, 3: false, 28: false, 14: true, 12: false, 2: false, 17: false, 27: false, 77: true, 82: false, 87: false, 97: false, 79: false, 32: false, 1: false, 4: true, 9: false, 19: false, 24: false, 26: false, 29: true, 34: false, 39: false, 44: false, 49: false, 54: false, 84: false, 109: false, 124: false, 8: false, 53: false, 23: false, 21: true, 6: false, 10: true, 20: false, 16: false, 18: true, 51: false, 55: true, 65: false, 70: false, 75: false, 121: true, 101: false, 106: false, 111: false, 123: false, 56: false, 41: false, 154: true, 152: true, 81: false, 31: false, 58: false, 103: false, 107: false, 42: false, 108: false, 43: false, 102: false, 57: false, 59: false, 104: false, 83: false, 33: false, 99: false, 86: false, 37: false, 63: false, 96: false, 69: false, 114: false, 50: false, 71: false, 47: false, 38: false, 88: false, 48: false, 67: false, 72: false, 68: false, 112: false, 89: false, 62: false, 113: false, 64: false, 73: false, 98: false, 61: false, 36: false, 74: false, 66: false, 40: false, 46: false, 138: false, 137: false, 139: false, 142: false, 140: false, 143: false, 136: false, 141: false, 216: false, 218: false, 217: false, 172: false, 203: false, 219: false, 173: false, 213: false, 174: false, 211: false, 170: false, 201: false, 215: false, 212: false, 171: false, 202: false, 204: false, 150: false, 168: false, 210: false, 166: false, 167: false, 169: false, 214: false, 200: false

From this, we can see that the German owns the fish, as 4 representing the German is true.

The time taken for our SAT Solver to solve Einstein's puzzle was 147ms with the heuristics implemented (All Clause heuristic and 1UIP conflict analysis heuristic).

Future Plans

In order to improve the effectiveness of the solver a few of the following ideas can be integrated into the solver as well.

Clause Learning

As the size of the formula increases so does the number of clauses being learnt. This may be the cause for a potential significant slow down of the algorithm. This is because with increase in the number of clauses learnt it also takes up valuable memory along with increase in processing power being used for analysis for the formula for various heuristics such as the 2 Clause Heuristic or All Clause Heuristic.

One way to mitigate this is to remove unnecessary or less useful clauses depending on various heuristics.To have a baseline comparator, implementations such as First In First Out with an Upper bound on the number of Clauses or a Random Removal of Clause on the clauses being learnt can be used.

To improve on this few of the heuristics that can be used to help in clause deletion are as follows

Clause Activity This heuristic has been used by many modern day Sat Solvers such as the MiniSat. This heuristic keeps a tab on the "recentness" of a clause being used. The more recently its been used, the more important the clause is. The key idea behind this being,a clause which has been used recently has a greater chance of being used again rather than a clause which has not been used in a long time which consumes extra memory and processing power.

Clause Size This heuristic involves the size of the clause. The smaller the clause the more important the clause is. The key idea behind this heuristic is that, a smaller clause has a higher restriction over the value a variable can take.

For example, a clause such as x3 forces the variable x3 to be true compared to a clause such as x3 $\vee$ x4.

Literal Block Distance (LBD) This heuristic involves counting the number of decision levels of the variables involved in the learning clause. The higher the number of decision levels the lesser its importance. The key idea behind this heuristic is that they glue together variables from the higher decision level to a variable from a lower (earlier) decision level, and the solver can then use this clause to set these variables earlier after backtracking.

Project 2 - Resolution Proof

For project 2, we decided to work on the third task - proof of unsatisfiability. We modified our SAT Solver from project 1 and extended it to also return the proof of unsatisfiability for only UNSAT problems. The proof of unsatisfiability works on the basis that if an empty clause can be derived from the clauses, then it must follow that the formula is unsatisfiable. Note that this proof of unsatisfiability only occurs when the conflict analysis heuristic is set to 1UIP instead of GRASP. In the class CDCL.java, in the following line - String solution = cdclSolver.solution("AllC", "1UIP", true); the last parameter can be changed from true to false, if you do not want the unsatisfiability proof to be outputted into resolutionProof.txt. The default setting is true, and hence, whenever a formula is UNSAT, the unsatisfiability proof would be outputted to resolutionProof.txt.

Approach

We kept a track of all the conflict clauses and the clauses they were being resolved with. This in turn created a pattern which we further pre-processed to suit the requirements of project 2. This works because resolution preserves satisfiability i.e. if a formula is satisfiable, then the clauses derived through resolution would also be satisfiable and vice versa for unsatisfiable formulas. An empty clause being derived is equivalent to a contradiction being found. One main thing to note is that we should not resolve and laern

existing clauses, as it would just increase the runtime.

Since our SAT Solver already determines whether a formula is UNSAT or not, for UNSAT formulas, we need to keep resolving until an empty clause is derived, because a empty clause being derived is always indicative of an UNSAT formula but it might take an extremely long time to derive an empty clause and might depend on the order of clauses we resolve. For example, let us look at the following formula: $(p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q)$

If we apply resolution taking the first 2 clauses as input, we can get p. Likewise, applying resolution on the 3rd and 4th clauses gives us $\neg$p. When we resolve p and $\neg$p, we derive an empty clause. Hence, we can conclude that the formula is unsatisfiable and this gives us the proof of satisfiability as well.

However, let us consider this approach, if we were to resolve clauses 1 and 4, i.e. $(p \vee q)$ and $(\neg p \vee \neg q)$, we would derive $q \vee \neg q$. And if we resolve clauses 2 and 3, i.e. $(p \vee \neg q)$ and $(\neg p \vee q)$, we would derive $\neg q \vee q$. Both of the derived clauses would always be true, as they are tautologies. It is important to note that this does not mean that the formula is satisfiable, just because the clauses derived from resolution are always true. It is therefore important to note that resolution simply causes your set of clauses to grow, and does not remove the 2 clauses that it resolved from. The set of clauses would expand and includes both original clauses in the formula and derived learnt clauses.

Analysis

For 42unsat.cnf, an UNSAT formula with 42 variables and 133 clauses, the time taken when the proof of unsatisfiability is required is 1459258 milliseconds, while the time taken when proof of unsatisfiability is not required is 28735 milliseconds.

Our proof of unsatisfiability by resolution took an exponentially long time especially on the larger formulas as we had to iterate through all the formulas again and again till we could derive an empty clause. The runtime also largely depended on the ordering of the formulas. In the aforementioned example, if we resolved clause 1 and 2 first and then clause 3 and 4, and then resolved the 2 newly derived clauses, we would immediately reach an empty clause. This only required the application of the resolution rule 3 times. However, if the ordering of the formula caused clause 1 and 4 to be resolved first and then clause 2 and 3, it would take a longer time and more steps of the resolution rule to derive the empty clause. Hence, the order of how we resolve the clauses plays an important role in how fast or slow an empty clause can be derived depends on the formula, and there is a factor of 'luck' involved, with our implementation which did not involve any special heuristics other than those we implemented for Project 1.

Heuristics for Future Plans

However, for future plans, we would try to work around this and resolve clauses in an order such that an empty clause can be derived faster. One such rule we could follow would be to resolve 2 clauses with only 1 literal different first, as opposed to resolving 2 clauses with more than 1 literals being different. In short, we would try to find clauses with minimal number of literals being different first, and resolve those first. This would hopefully speed up the proof of unsatisfiability, though the overhead required to calculate the difference in literals between 2 clauses for all of the clauses may take up a long time, especially for larger formulas.

Reflections

Mukesh

Personally for me the whole project although difficult was really interesting and exciting to implement. Although I initially started out thinking I had a clear idea of what I wanted to implement, as time passed by and I started to implement, the more I implemented the more I realised how confused I was.

This gave me a clearer perspective on how things really change when implementing ideas we have and also showed me the importance of valuable skills like researching, trying various implementations and patience.

Through this project, I really learnt the implementation of CDCL algorithm, the importance of heuristics and why such Sat Solvers are useful in general. I also learnt how little I knew about the current state of art implementations and acquired new level respect for them. Apart from the hard skills I also learnt several soft skills such as the importance of testing, backups, being clear with requirements, communication with teammate, work division and most importantly being able to predict and judge the time needed to implement a feature or a tweak.

Personally I feel I would grade myself A- for this work as there are more heuristics such as Random restarts,clause removal etc. I could have implemented with better planning and time management.

Hyma

This project was undeniably very challenging and required a lot of time and effort. In my opinion, the toughest part was encoding Einstein's puzzle, and this can be attributed to the sheer complexity and amount of clauses required to encode the entire puzzle. This really made me realise how difficult it would be to actually model real world problems, as just a few lines of rules and hints required a lot of effort to model accurately. While encoding the puzzle, I initially wanted to optimise it by having as little rules as possible, but this proved very difficult without compromising on the accuracy and completeness. This made me more interested in learning how real world problems are modelled at SAT problems and I wanted to know more about the practical

applications of what I was learning, and learning the concepts and undergoing this project really made me have a greater appreciation for the practical applications.

The most insightful takeaway was the importance of 'getting your hands dirty', as Professor Kuldeep always says. He stresses the need to try solving it on your own first, instead of being spoon fed the answer. When I first looked at the project, I thought it would be just an implementation of what we learnt in class and using our prior knowledge of propositional logic to create the SAT Solver. However, as we started coding, we realised the number of gaps in our knowledge and understanding, that caused us to initially implement the CDCL algorithm wrongly and only realise it later on while we were trying to debug. This really made me understand the importance of fully understanding and internalising the concepts.

I would grade myself an A- for this project as I feel that I have invested a lot of time and effort into this project and my understanding was really firmed up. However, if I had cleared my misconceptions right at the start itself, I could have been more efficient with my time and been able to implement the various clause learning techniques listed in the section 'Future Plans', and also optimise the unsatisfiability proof by resolving the clauses in an optimal order.

References

[1] Marques-Silva, J., Lynce, I., Malik, S. (2008). Chapter 4: Conflict-Driven Clause Learning SAT Solvers, Handbook of Satisfiability, IOS Press.

[2] Hoeovsk, M. (2019). Modern SAT solvers: fast, neat and underused (part 3 of N), The Coding Nest, retrieved from https://codingnest.com/modern-sat-solvers-fast-neat-and-underused-part-3-of-n/

[3] Nabeshima1, H., Inoue, K. (2017) Coverage-Based Clause Reduction Heuristics for CDCL Solvers, Springer International Publishing, SAT 2017, LNCS 10491, pp. 136144. doi: 10.1007/978-3-319-66263-3 9.

[4] Soos M., Nohl K., Castelluccia C. (2009) Extending SAT Solvers to Cryptographic Problems. In: Kullmann O. (eds) Theory and Applications of Satisfiability Testing - SAT 2009. SAT 2009. Lecture Notes in Computer Science, vol 5584. Springer, Berlin, Heidelberg. doi: https://doi.org/10.1007/978-3-642-02777-2_24

[5] Cryptominisat solver installed and retrieved from https://github.com/msoos/cryptominisat. Citation of paper in [4].

[6] Lauria, M., Vinyals, M., Elffers, J., Mika, M., Nordstrm, J. CNFgen: Combinatorial benchmarks for SAT Solvers. Retrieved and installed from http://massimolauria.net/cnfgen/.

[7] Example SAT formulas retrieved from: SATLIB - Benchmark Problems (2011), https://www.cs.ubc.ca/ hoos/SATLIB/benchm.html

[8] Zhang, L., Madigan, C. F., Moskewicz, M. H., Malik, S. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. Retrieved from http://math.ucsd.edu/ sbuss/-CourseWeb/Math268_2007WS/ zhang01efficient.pdf

[9] Torlak, E. CSE507 Computer-Aided Reasoning for Software. Retrieved from https://courses.cs.washington.edu/courses/cse507/14au/slides/L3.pdf