

## Midterm 1: SAT Solver

### Introduction:

The purpose of this midterm was to create a functional SAT Solver in the programming language of our choice. The goal was to create a toolset capable of taking an SoP boolean function as an input, and recursively assigning decisions and deductions to prove whether or not the function was satisfiable. This report outlines the methods used to create the system, the role of each team member, and examples of the tool execution.

### Summary:

As stated in the introduction, the goal of this midterm was to create a SAT Solver similar to other DPLL SAT Solvers such as MiniSAT. Due to the large scope of this project and the limited coding experience of our team, we decided to implement our SAT Solver using python, as the syntax is much simpler than other programming languages. This project consisted of three main objectives, the SAT Solver, the SoP to CNF algorithm, and the XOR equality test. Before discussing the role of each team member, let's discuss the algorithms and methods used in each objective.

### *SAT Solver*

To begin the project, we focused our attention on the SAT Solver. After looking at the scope of the project, we decided that the best course of action was to create a working SAT Solver before attempting to implement the SoP to CNF algorithm or the XOR equality test. Therefore, we began by creating a SAT Solver capable of identifying the satisfiability of a system given a CNF file.

Our SAT Solver is a backtracking-based system based on the Davis-Putnam-Logemann-Loveland (DPLL) search algorithm. As we learned in the lecture, the DPLL algorithm works by assigning a truth value to a literal, simplifying the input formula, and then recursively checking if the simplified formula is satisfactory. In order to implement this algorithm into a python script, we made use of the following pseudo-code.

```

Algorithm DPLL
  Input: A set of clauses  $\Phi$ .
  Output: A truth value indicating whether  $\Phi$  is satisfiable.

function DPLL( $\Phi$ )
  while there is a unit clause  $\{l\}$  in  $\Phi$  do
     $\Phi \leftarrow \text{unit-propagate}(l, \Phi)$ ;
  while there is a literal  $l$  that occurs pure in  $\Phi$  do
     $\Phi \leftarrow \text{pure-literal-assign}(l, \Phi)$ ;
  if  $\Phi$  is empty then
    return true;
  if  $\Phi$  contains an empty clause then
    return false;
   $l \leftarrow \text{choose-literal}(\Phi)$ ;
  return DPLL( $\Phi \wedge \{l\}$ ) or DPLL( $\Phi \wedge \{\text{not}(l)\}$ );

```

*Figure 1: DPLL Algorithm Pseudocode [1]*

As can be seen from Figure 1, the DPLL Algorithm consists of three main blocks of code. These blocks are the unit propagation, pure literal elimination, and DPLL recursion. The unit propagation code checks the input function for any clauses containing only one literal (unit clauses) and assigns the necessary value to that literal to make it true. The code then removes every clause containing that literal, and every instance of that literal's complement. This code was implemented in a method called "unit\_clause(cnf)" in the "sat.py" file.

After the unit propagation code is finished, we move on to the pure literal elimination. This portion of code searches for any literal that exists in the function as always positive or always negative. This is known as a pure literal. If a literal only exists as positive or negative, it is easy to assign the necessary value to satisfy the clauses that contain that literal. The method called "pure(cnf, pos, neg)" searches for any pure literal in the CNF, assigns the necessary value to satisfy the literal, and removes all satisfied clauses from the CNF array.

After all pure literals are removed from the CNF, the system checks to see if an empty clause or function exists. If the entire function is empty, the code returns "true" because an empty function is always satisfiable. If the function contains an empty clause, the code returns false, because there is no way to satisfy an empty clause. This portion of code can be seen in the "dpll(cnf)" method.

Once the code checks for an empty function or clause, the next step is to assign a truth value to a literal in the function. This portion of the code occurs in the same for loop as the empty clause check. If the function contains no empty clauses, each literal in the function is added to a new sorted literal array called "newLit". From there we create two copies of the

function called “posCopy” and “negCopy”. This allows us to save the current function without it being overwritten by the recursion. We then assign a truth value to the smallest literal in the “newLit” array and send that value into each copy of our function respectively. Finally, we recursively call the “dpll(cnf)” method with our new function and the process repeats itself. If the algorithm finds a set of literals that satisfies the function, the code will return “SAT” along with the variable assignments that cause the system to be satisfied. Then it will append the inverse of that literal assignment to the end of the “out.cnf” file and search for another solution to the system. This will continue until all possible satisfiable input combinations are found. These combinations are stored in a file called “out.txt”. However, if the algorithm cannot find a set of literals that satisfies the function, the code will return “UNSAT”. Examples of both satisfiable and unsatisfiable functions will be shown in a later portion of this report.

### SoP to CNF Algorithm

For the second part of the midterm, the team decided to try and work on the implementation of the CNF input going into our SAT Solver. Since we know that our SAT Solver is working with sample CNF files, we now wanted to find a way to take a boolean expression in the form of an SoP and change it to a CNF file.

The first thing we wanted to do was implement the different gates that we would need to define for our CNF converter to be operable. The three gates that we needed to implement were AND, OR, and NOT gates. In order to construct these gates, we can use the formulas seen in class to convert SoP to PoS. These formulas can be seen in Figure 2 and were used as references for the code we created. One example of how we implemented these formulas can be seen below in Figure 3, where I show how we implemented the NOT gate into our code. The not\_gate method takes an input (such as  $\sim x1$ ) and has an output of x2. We first split the input by the “~” and then we can return the following result seen below.

$z = (x)$ <p>(yes this is just a wire)</p> $[\bar{x} + z][x + \bar{z}]$	$z = \text{NOR}(x1, x2, \dots, xn)$ $\left[ \prod_{i=1}^n (xi + z) \right] \left[ \left( \sum_{i=1}^n xi \right) + z \right]$	$z = \text{OR}(x1, x2, \dots, xn)$ $\left[ \prod_{i=1}^n (\bar{xi} + z) \right] \left[ \left( \sum_{i=1}^n xi \right) + z \right]$
$z = \text{NOT}(x)$ $[x + z][\bar{x} + \bar{z}]$	$z = \text{NAND}(x1, x2, \dots, xn)$ $\left[ \prod_{i=1}^n (xi + z) \right] \left[ \left( \sum_{i=1}^n \bar{xi} \right) + z \right]$	$z = \text{AND}(x1, x2, \dots, xn)$ $\left[ \prod_{i=1}^n (xi + z) \right] \left[ \left( \sum_{i=1}^n xi \right) + z \right]$

Figure 2: Formulas for all Logic Gates from ECE 480 Lecture

```
# NOT gate method based on NOT formula from lecture
def not_gate(f, out):
    func = f.split("~")
    print()
    print("NOT GATE EXAMPLE: f = '~x1' and out = 'x2'")
    print(func)
    print("-" + func[1][1:] + " -" + out[1:] + " 0\n")
    print(func[1][1:] + " " + out[1:] + " 0\n")
    return "-" + func[1][1:] + " -" + out[1:] + " 0\n" + func[1][1:] + " " + out[1:] + " 0\n"
```

```
NOT GATE EXAMPLE: f = '~x1' and out = 'x2'
['', 'x1']
-1 -2 0

1 2 0
```

Figure 3: Functional NOT Gate example

After the completion of proving all the formulas for AND, OR, and NOT gates in our “cnf.py” file, it was now time to build our CNF. First, since we instruct the user to put a function into our solver in the form of an SoP boolean expression, we take that string input with acceptable input variables and immediately split the function given in the string by its OR’s. We split the expression by its OR gates because we want to create a new array for AND gate clauses. Then, we want to split the array we just created again into a smaller nested array of literals by splitting the AND’s. Then, when we see a “~” anywhere within the array, we want to call the NOT gate method explained above to append to the output and replace all “~” inputs with new outputs. We keep track of the new outputs using the var\_count method. This method searches the array for the highest valued literal and adds 1 to that number to be used as a new output. These steps can be seen in Figure 4 with a specific example that we used for the majority of our debugging and testing.

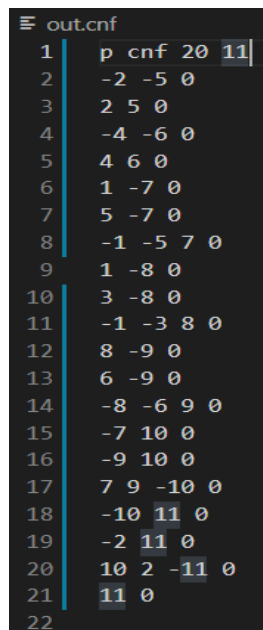
```
PS C:\Users\Nick\Desktop\SAT SOLVER\ECE480_Midterm-5> python cnf.py x1.~x2+x1.x3.~x4+x2
split at OR gates
['x1.~x2', 'x1.x3.~x4', 'x2']
-----
split at AND gates
[['x1', '~x2'], ['x1', 'x3', '~x4'], ['x2']]
-----
NOT gates removed
[['x1', 'x5'], ['x1', 'x3', 'x6'], ['x2']]
-----
```

Figure 4: Example to show how we split OR, AND, and NOT gates

Once this part of the code was complete, we then needed to find a way to apply our AND, OR, and NOT functions to all remaining literals stored in our array. To do this, we first looked inside each AND gate and reviewed the size of the array. We then applied the AND function to

the first two literals (from left to right) within the array. This would insert a new output of the AND-gate function in the place of the two inputs we just ANDed. If the array is longer than two literals, we will continue to run the AND-gate function until all arrays are complete and all we are left with is one literal in each array. Once we get to this point, we can finally call the OR-gate function on the remaining literals and create one single (and the highest) literal which represents the function's output.

Finally, we are able to fully create and call these new literals to a new cnf file that we brilliantly called “out.cnf”. This file is called in the main section of the SAT code and will always update based on the user’s input. For example, we can observe in Figure 5 below our simple example of  $F = x1.\sim x2+x'.x3.\sim x4+x2$  being turned into a CNF file. Much like homework 3, we decided to also print at the top of the CNF file, the length of the file (20), as well as the output value (11).



```

1  p cnf 20 11
2  -2 -5 0
3  2 5 0
4  -4 -6 0
5  4 6 0
6  1 -7 0
7  5 -7 0
8  -1 -5 7 0
9  1 -8 0
10 3 -8 0
11 -1 -3 8 0
12 8 -9 0
13 6 -9 0
14 -8 -6 9 0
15 -7 10 0
16 -9 10 0
17 7 9 -10 0
18 -10 11 0
19 -2 11 0
20 10 2 -11 0
21 11 0
22

```

Figure 5: CNF File example

### ***XOR Implementation:***

For this section of the project, we unfortunately ran out of time and could not fully implement the XOR gate to compare two input boolean functions. However, if we did have the time to construct such a feat, we would start by allowing our CNF file to take in 2 functions. We could do this by creating another variable that asks for a second argument with the sys.argv command. Next, within our cnf.py code, we would have to create another method for the XOR algorithm which can be seen in Figure 6. This algorithm would be run consecutively of the normal SAT and CNF implementations explained above. We would run each function on its own and then, send the output of each sat.py function into the input of the XOR method. This would lead to us creating another individual file that would need to be run through the SAT once again. Once that is all complete, we could easily see if the two functions were equal to each other and Satisfiable.







Gate Type	Gate Equation	CNF Expression
	$z = NOT(x)$	$(x + z) \cdot (\bar{x} + \bar{z})$
	$z = NOR(x, y)$	$(\bar{x} + \bar{z}) \cdot (\bar{y} + \bar{z}) \cdot (x + y + z)$
	$z = NAND(x, y)$	$(x + z) \cdot (y + z) \cdot (\bar{x} + \bar{y} + \bar{z})$
	$z = AND(x, y)$	$(x + \bar{z}) \cdot (y + \bar{z}) \cdot (\bar{x} + \bar{y} + z)$
	$z = OR(x, y)$	$(\bar{x} + z) \cdot (\bar{y} + z) \cdot (x + y + \bar{z})$
	$z = XOR(x, y)$	$(\bar{x} + y + z) \cdot (x + \bar{y} + z) \cdot (\bar{x} + \bar{y} + \bar{z}) \cdot (x + y + \bar{z})$

Figure 6: CNF gate Equation Table [2]

**Team Information:****Roles**

Nicholas Leeseberg:

- Create CNF converter
- Research python

Jake Nash

- Create SAT Solver,
- Research python

Collectively:

- We both helped debug and implement cnf.py and sat.py together

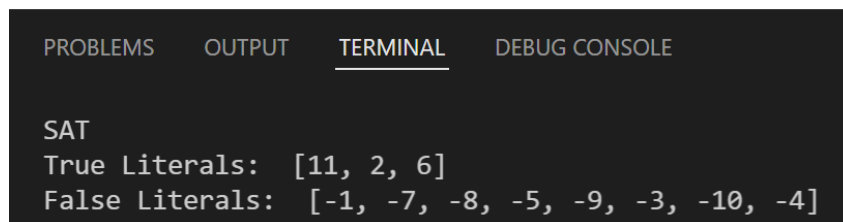
**Time Spent on Project**

- 50+ Hours
- Met everyday Monday, March 7, 2022 - Sunday, March 13, 2022

**Tool Execution Examples:**

In order to test the functionality of our system, we tested a series of satisfiable SoP functions and unsatisfiable SoP functions to ensure that our CNF algorithm produced the correct output and our SAT Solver correctly identified the satisfiability of each function. After running each function through our code, we ran the same function through pyEDA in order to ensure that our system outputs the correct satisfiability, as well as the correct inputs to cause the system to be satisfiable. Below we have shown two examples of this process, one satisfiable function, and one unsatisfiable function.

SAT EXAMPLE:  $f = x_1 \cdot \sim x_2 + x_1 \cdot x_3 \cdot \sim x_4 + x_2$



```

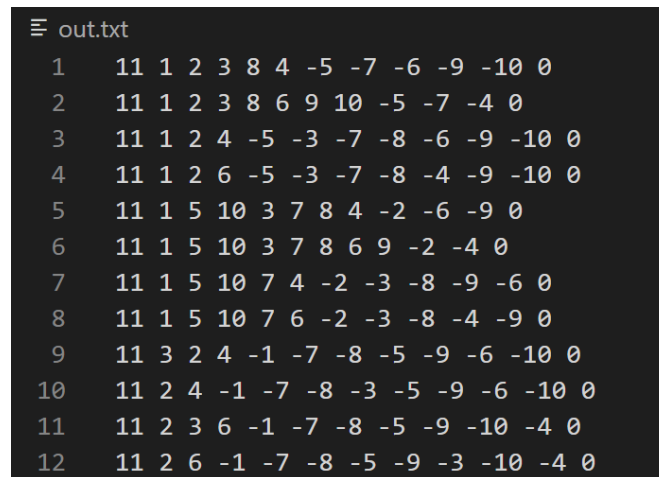
PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE

SAT
True Literals:  [11, 2, 6]
False Literals: [-1, -7, -8, -5, -9, -3, -10, -4]

```

*Figure 7: Terminal Output of sat.py for SAT Example Function*

Figure 7 shows one iteration of the terminal output of sat.py with the input function shown above. This figure only shows one possible set of literals for the system. However, the most important aspect to note is the line “SAT”, which signifies that the system is satisfiable. All possible sets of literal for the system are shown in Figure 8.



```

≡ out.txt
1  11 1 2 3 8 4 -5 -7 -6 -9 -10 0
2  11 1 2 3 8 6 9 10 -5 -7 -4 0
3  11 1 2 4 -5 -3 -7 -8 -6 -9 -10 0
4  11 1 2 6 -5 -3 -7 -8 -4 -9 -10 0
5  11 1 5 10 3 7 8 4 -2 -6 -9 0
6  11 1 5 10 3 7 8 6 9 -2 -4 0
7  11 1 5 10 7 4 -2 -3 -8 -9 -6 0
8  11 1 5 10 7 6 -2 -3 -8 -4 -9 0
9  11 3 2 4 -1 -7 -8 -5 -9 -6 -10 0
10 11 2 4 -1 -7 -8 -3 -5 -9 -6 -10 0
11 11 2 3 6 -1 -7 -8 -5 -9 -10 -4 0
12 11 2 6 -1 -7 -8 -5 -9 -3 -10 -4 0

```

*Figure 8: Output File of sat.py for SAT Example Function*

Figure 8 shows the output file of our SAT Solver with the input function shown above. As can be seen from Figure 8, this function has a total of 12 unique literal assignments which make the system satisfiable. In order to ensure that our code was working properly, we plugged the same function into pyEDA to verify its satisfiability. That code can be seen below.

```
In [11]: ▶ import pyeda
          from pyeda.inter import *

In [30]: ▶ a, b, c, d, e = map(bddvar, 'abcde')
          f = a & ~b | a & c & ~d | b
          list(f.satisfy_all())

Out[30]: [{a: 0, b: 1}, {a: 1}]
```

*Figure 9: pyEDA Code for SAT Example Function*

Figure 9 shows the pyEDA code used to verify the output of our code. The first important thing to note is that the literals are named a, b, c, d, e rather than x1, x2, x3, x4, x5. This is because the pyEDA does not like outputting literals with numbers in their name. Therefore, we will consider  $a=x_1$ ,  $b=x_2$ , and so on. The next important thing to note is the output of the code. As can be seen from Figure 9, pyEDA's output looks nothing like the output of our code in Figure 8. However, these two outputs are conveying the same information. The output in Figure 9 states that the function is satisfiable whenever  $a=x_1=0$  and  $b=x_2=1$ , or when  $a=x_1=1$ . Looking at the output in Figure 8, we can see that of the 12 possible literal assignments, eight of them have  $x_1=1$  and four of them have  $x_1=0$ . The eight assignments with  $x_1=1$  prove that when  $x_1=1$ ,  $x_2$ ,  $x_3$ , and  $x_4$  can be anything. This is the same information as “{a: 1}” in Figure 9. As for the four assignments that have  $x_1=0$ , all four of these assignments also have  $x_2=1$ , which is the same as “{a: 0, b: 1}” in Figure 9. Therefore, because our output is consistent with pyEDA's output, we can say that this test case was successful.

UNSAT EXAMPLE:  $f = x_1 \sim x_1$

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
-----
['p cnf 6 3\n', '-1 -2 0\n1 2 0\n', '1 -3 0\n2 -3 0\n-1 -2 3 0\n', '3 0\n']
PS C:\Users\13039\Desktop\ECE 480\Midterm_1\ECE480_Midterm> python sat.py
UNSAT
PS C:\Users\13039\Desktop\ECE 480\Midterm_1\ECE480_Midterm> □
```

*Figure 10: Terminal Output of sat.py for UNSAT Example Function*

Figure 10 shows the terminal output for the UNSAT example function shown above. When the system is unsatisfiable. Because there are no possible solutions to the system, our code simply returns “UNSAT” in the terminal, with output.txt being empty.



```
In [11]: ▶ import pyeda
          from pyeda.inter import *

In [33]: ▶ a, b = map(bddvar, 'ab')
          f = a & ~a
          list(f.satisfy_all())

Out[33]: []
```

*Figure 11: pyEDA Code for UNSAT Example Function*

Figure 11 shows the code and output of the pyEDA implementation of the unsatisfiable function shown above. As in the last example, the variable 'a' represents literal 'x1' in our code. As can be seen in Figure 11, the output is an empty array. This is because the system is unsatisfiable, and therefore has no possible solutions. Due to this output, we can conclude that our code works for unsatisfiable functions as well as satisfiable functions.

### **Conclusion:**

The purpose of this midterm was to create a SAT Solver based on the famous DPLL backtracking algorithm. The system had to be able to take an SoP boolean function as an input, and output whether or not the system was satisfiable, all possible input combinations that make the system satisfiable, and the smallest set of input variables that make the system satisfiable. On top of that, the code had to be able to compare two functions and find input variables that could cause the functions to be unequal. Due to the large scope of this project and our limited coding experience, we put many hours into this project and were unable to complete every aspect of the assignment. However, we were successfully able to create a SAT Solver that takes an SoP function as an input and outputs all possible input combinations which satisfy the equation. We were unable to implement the XOR function equality test or the smallest set of input variables because we ran out of time. However, given the amount of time we put into this project and our success despite our limited coding experience, we are extremely proud of what we were able to accomplish.

Works Cited

- [1] “DPLL Algorithm.” *Wikipedia*, Wikimedia Foundation, 9 Mar. 2022, [https://en.wikipedia.org/wiki/DPLL\\_algorithm](https://en.wikipedia.org/wiki/DPLL_algorithm).
- [2] Sagahyroon, Assim, and Fadi A. Aloul. “Using Sat-Based Techniques in Power Estimation.” *Microelectronics Journal*, Elsevier, 27 June 2007, <https://www.sciencedirect.com/science/article/abs/pii/S0026269207000948>.
- [3] YibaiMeng. “Yibaimeng/DPLL-SAT-Solver: A Naive Implementation of DPLL SAT Solver in Python.” *GitHub*, <https://github.com/YibaiMeng/DPLL-SAT-solver>.