Project Readme Template

Version 1 9/11/24

A single copy of this template should be filled out and submitted with each project submission, regardless of the number of students on the team. It should have the name readme_"teamname"

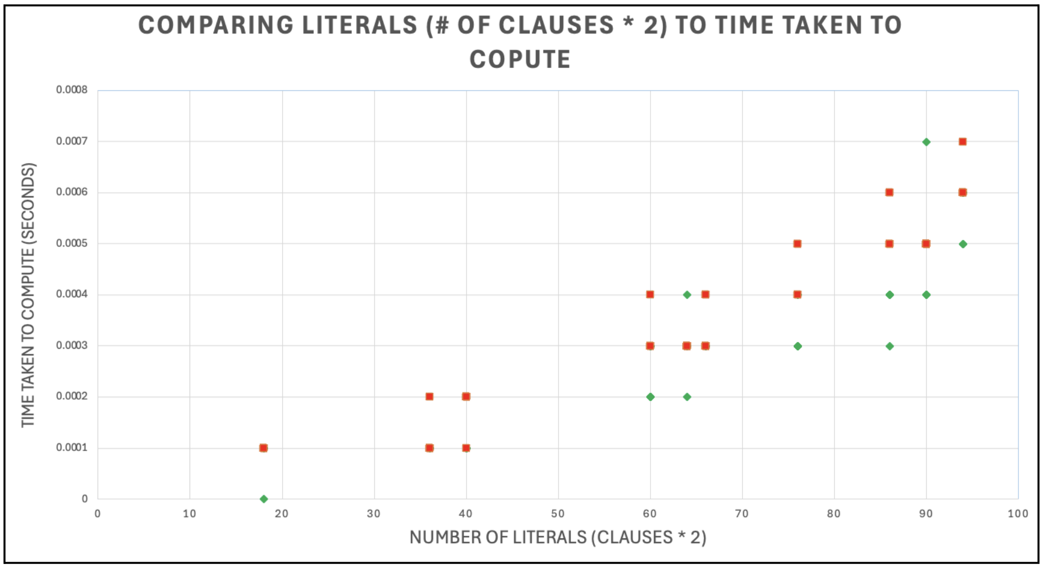
Also change the title of this template to "Project x Readme Team xxx"

1	**Team Name:** oheldrin
2	**Team members names and netids:** Olivia Heldring, oheldrin
3	**Overall project attempted, with sub-projects:** Implementing a polynomial time 2-SAT solver with the DPLL algorithm
4	**Overall success of the project:** I think my project was quite successful! My code works, and I understand everything I did! A more detailed explanation is below.
5	**Approximately total time (in hours) to complete:** 12
6	**Link to github repository:** **https://github.com/oliviaheldring/TheoryProject_oheldrin**
7	List of included files (if you have many files of a certain type, such as test files of different sizes, list just the folder): (Add more rows as necessary). Add more rows as necessary.

File/folder Name	**File Contents and Use**
Code Files	
twoSATcode_oheldrin.py	This contains all of my functions, including my main function. This file reads in the input data, processes each problem, and lists out key information like Problem Number, Number of Clauses, Number of variables, Number of Literals, the Clauses themselves, Satisfiability, and Time taken.
Test Files	
input_file_oheldrin	This was in Canvas and given by Kogge. It contains 100 2-SAT problems.
Output Files	
output_file_oheldrin.txt	The results of my code are printed to the terminal and

	stored in this output file. In this file, you will find the following information about all 100 problems: Problem Number, Number of Clauses, Number of variables, Number of Literals, the Clauses themselves, Satisfiability, and Time taken.

Plots (as needed)



- **This graph compares the number of literals (clauses * 2) to the time it takes to compute. As we can see, the time taken grows exponentially as the number of literals grows. The red squares represent unsatisfiable, and the green diamonds represent satisfiable.**

Other	In my repo, I also attached my **readme**, my **teamwork file**, and a file about all my data represented as a clear table. The table/data file is not required at all. It was just the excel sheet I used to generate my plot.

8	**Programming languages used, and associated libraries:** ● Python ● Libraries: time

9	**Key data structures (for each sub-project):** 1. **List:** • The **problems** list holds multiple CNF problems (tuples) from the input file • **current_problem** temporarily holds the clauses for the current problem being processed. • **Lines** is a list of strings used for processing the input file • **Parts** is a list of strings separated by white space in processing input. • **clauses** is a list of lists. • **literals** is a list that flattens all the clauses, which is then used for pure literal elimination. • **unit_clauses** is a list of clauses with only one literal, also used for pure literal elimination. 2. **Set** • The **assignment** set keeps track of which variables are assigned a truth value. This is used in the unit propagation and pure literal elimination functions. 3. **Integer** • **problem_number** • **Num_clauses** • **num_variables**
10	**General operation of code (for each subproject)** My code implements a polynomial time 2-SAT solver using the DPLL algorithm: 1. **Input Reading (read_cnf function):** ○ This function reads a CNF input file that holds multiple 2-SAT problems. ○ This function captures the problem number, number of clauses, number of variables, and develops a list of all the problem content information. 2. **Unit Propagation (unit_propagation function):** ○ If a clause contains a single literal (unit clause), the algorithm assigns that literal a value to satisfy the clause. 3. **Pure Literal Elimination (pure_literal_elimination function):** ○ If a literal appears in the formula without its negation (only one polarity), it

	can be assigned a value to simplify the formula. 4. **Simplification (simplify function):** ○ This function simplifies the formula by removing clauses that are satisfied and adjusting the remaining ones. 5. **Satisfiability Check (is_satisfiable function):** ○ This checks if the problem is satisfiable or unsatisfiable by seeing if the clauses have been fully reduced. 6. **DPLL Algorithm (dpll function):** ○ This is the core recursive function. ○ It applies unit propagation and pure literal elimination to simplify. ○ If the problem is not yet solvable or unsolvable, it guesses an assignment and recurses with both possible truth values. 7. **Main Function (__main__):** ○ The main function handles file input and output. ○ It reads the CNF problems, solves each problem, and writes the results to the terminal and to an output file.
11	**What test cases you used/added, why you used them, what did they tell you about the correctness of your code.** - First, ***I did a couple of Kogge's input file problems by hand*** and ensured my solutions matched my code's output. For this, I solved Kogge's problems 1000, 1001, and 1002 to ensure my solution matched my code's. He said this was sufficient to ensure with these three current, all 100 would be correct too. The input file in my github repo has all 100 problems. - Next, I **googled three 2-SAT problems/solutions online and fed them to my code**. Again, my code matched the online solution to these sample problems! - This redundant correctness gave me confidence that the code was working properly, and that a solution for any number of literals for a 2-SAT problem would be correct.

12	**How you managed the code development** - I focused on breaking the problem into clear, manageable steps. I started with reading and processing the input file, making sure I could properly extract and organize the clauses for each problem. This was key for making sure I had the right information—like the number of variables and clauses—before jumping into the solution process. - From there, I worked on the core functions of the DPLL algorithm. I wrote the unit propagation and pure literal elimination functions first, as these steps reduce the problem size and can sometimes solve the problem outright. The DPLL function was simple but recursive, so I spent some time checking my process there to ensure literals were being eliminated properly. Finally, I added timing mechanisms to track performance and compared that on a scatter plot to the problem complexity (number of literals).
13	**Detailed discussion of results:** - The graph of these results shows that as the number of literals (clauses * 2) grows, then the average time to compute goes up exponentially. This makes sense, as more literals means the possibility of way more assignment possibilities to check in the DPLL calculator. I was happy with my results, because the satisfiability are all correct and the time taken per problem is a reasonable reflection of the complexity level! - The graph attached in my repo depicts this. Note that red dots represent unsatisfiable, and green dots represent satisfiable.
14	**How team was organized** I worked alone. Everything submitted was my own.
15	**What you might do differently if you did the project again** - I would write this code in VSCode as opposed to the terminal. When using VSCode, you can step through each variable and see where/which line the integers change. For a while, my code was not properly capturing the “Problem Number” (ie. 1000, 1001, etc). Had I been using VSCode, I could have very quickly realized on which specific line the problem number was getting erased, and edited it immediately. Instead, I worked in the Terminal and had to throw a bunch of print statements everywhere.