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: **ariter**
2	Team members names and netids: **Alyssa Riter - ariter**
3	Overall project attempted, with sub-projects: **Implementing a polynomial time 2SAT solver (DPLL algorithm)**
4	Overall success of the project: **The project was successful in determining whether the randomly generated 2SAT CNFs were satisfiable or unsatisfiable. The algorithm successfully shows a graph that begins to represent exponential worst-case time complexity. As observed by my graph, as the complexity of the problem increased (shown by the number of variables) the execution time of the problem also increased quickly. We can also observe that the unsatisfiable CNFs took longer to execute than the satisfiable CNFs as is characteristic of NP-complete problems. Additionally, this program introduced me to the DPLL algorithm and what it is like to implement this algorithm in a program. The DPLL algorithm showed me how a tool can be used to solve SAT, which is NP-complete. I was able to successfully incorporate backtracking, unit propagation, and pure literal elimination through a python program. It was a great learning opportunity to further understand NP-complete problems while also reviewing essential coding skills.**
5	Approximately total time (in hours) to complete: **6-7 hours**
6	Link to github repository: https://github.com/AlyssaRiter23/Theory-DPLLSolver/tree/main
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	
dpll_ariter.py	**This code implements the DPLL algorithm. It also generates CNFs to test the 2SAT problem as well as code to graph the results of the program.**
Test Files	
dpll_ariter.py	**Randomly generated CNFs in the dpll_ariter.py**

check_ariter.txt	**program using the *generate_random_CNFs* function. The randomly generated CNFs can be found in the check_ariter.txt file.**
Output Files	
check_ariter.txt output_ariter.txt	**The check_ariter.txt file contains the CNFs generated from the *generate_random_CNFs* function in the dpll_ariter.py program which can be considered both a test file and an output file. output_ariter.txt contains the literal assignments that were created and whether or not each CNF was satisfiable or unsatisfiable, it also records the execution time for each trial.**
Plots (as needed)	
execution_time_plot.png	**This image shows the result of the DPLL algorithm and the polynomial time execution. This plot was created in a function called *plot_execution_times* in the dpll_ariter.py program.**

8	Programming languages used, and associated libraries: **Programming Language: Python3** **Associated libraries: time, matplotlib, numpy, random, copy**
9	Key data structures (for each sub-project): **Dictionaries (for the true/false assignments of the literals), Lists (for CNFs and to keep track of execution times), and Tuples (for random generation of clauses for CNFs)**
10	General operation of code (for each subproject): **The program starts with the function *test_dpll_polynomial* and we begin by generating random 2SAT CNFs formulas. For the number of clauses I set it to be a random number between the number of variables and three times the number of variables so that there was a fairly random and substantial amount of cases to test. I ensured that what was generated were clauses with 2 literals in it. I then used python's random number generator to give the literal a random number and negation assignment. I also avoided putting any duplicate clauses in the CNF. I wrote these randomly generated clauses to a file so that I could manually solve them and verify my program's correctness. Next, I used python's time library to calculate the execution time of the problem. I separate the results of the DPLL algorithm into satisfiable and unsatisfiable lists to differentiate when plotting. In the *DPLLAlgorithm* function I pass in the random CNF that was generated and a dictionary for the assignments to be placed as arguments. I start by testing if there is a unit clause in the program which would automatically have to be true considering that the only way for the whole CNF to be true is if the clause is also true. If there is an empty clause in the CNF I immediately return false as this is a base case. Next, the DPLL algorithm function checks for pure literals- a literal is pure if its negation is not found in the CNF - I set this literal to true and check if**

	there is an empty clause. If the CNF has no clauses we return true and if it contains no literals we return false. To improve the efficiency of the algorithm I choose the literals based on how often they appear in the CNF (I pick the most frequent one first). I then have two recursive calls to the *DPLLAlgorithm* function. The first is with the unit propagation of the literal and an assignment of true, and if that is unsatisfiable I test the unit propagation with the negation of the literal and an assignment of false. I return the satisfiability of the CNF and the assignment dictionary from the function. Lastly, I plot the results of the algorithm using python's scatter plot from matplotlib.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code: **The test cases that I used were generated in a function in my program. They were random generations of 2SAT CNFs that allowed me to ensure that my program was robust and could handle a variety of cases. I ranged the number of variables in the problem from 3 to 150 to show a great increase in the complexity of the problem and demonstrate the fundamental characteristic of the NP-complete problem. I also performed 10 trials for each number of variables and varied the number of clauses in each trial. This allowed me to validate the robustness of my program by seeing it successfully parse a vast number of random CNFs. I verified the correctness of my code by printing all of the generated 2SAT CNFs to a file as well as the assignments that my program outputted for each literal to ensure that the assignments it created actually satisfied the CNF. For the unsatisfiable CNFs I ensured that they were unsatisfiable by manually checking the output.**
12	How you managed the code development: **I used VScode and Github to manage my code development. I used the algorithm and pseudocode provided by Wikipedia to fully understand the algorithm before implementing it into my code. I created numerous functions in my code to improve readability and make debugging easier.**
13	Detailed discussion of results: **The DPLL algorithm works to improve normal 2SAT solvers by incorporating unit propagation and pure literal elimination along with the usual backtracking. As shown in my program, unit propagation consists of removing every clause containing a unit clause's literal and also removing the complement of a unit clause's literal from every clause that contains that complement. Pure literal elimination is shown in the program by assigning every literal that occurs with only one polarity in the CNF to true. As can be seen in the plot that was outputted by the program, the execution time of the program increases as the complexity of the problem increases, and it does so in a manner that gives us insight into the exponential worst-case time complexity. We can also observe that the execution time of the unsatisfiable CNFs are greater than that of the satisfiable CNFs which is in accordance with the foundation of NP-complete problems. As shown by the graph for NP-complete problems there is no way to find a solution quickly. Once a solution is found we may be able to verify it fairly quickly (as I did to verify the correctness of my program) but the actual time required to solve the problem using the DPLL algorithm increases rapidly as the size of the problem grows. While the DPLL algorithm is efficient compared to brute force, it still has an exponential worst-case time complexity, which is expected since SAT is NP-complete meaning that no known algorithm, including**

	DPLL, can solve all instances of SAT is polynomial time.
14	How the team was organized: **There was no team as I completed this project individually. However, I think that overall the use of Github was greatly beneficial for both an individual and team because it provides a way to organize all the programs in a clear and concise manner. Additionally, members can contribute and view each other's work, allowing for good collaboration. As an individual, Github made it easier to organize and layout all the necessary programs and files that I need for submission. I was responsible for writing the functions to implement the DPLL algorithm, as well as a function to generate random test cases, and a function to graph the execution times.**
15	What you might do differently if you did the project again: **The number of loops and functions in the code is redundant and decreases efficiency. Overall, I think that the efficiency of the code could be improved through shirt circuit evaluations and less unnecessary while and for loops.**
16	Any additional material: **N/A**