Project 1 Readme Team aarnett2

Version 1 9/11/24

1	Team Name: aarnett2
2	Team members names and netids: Anna Arnett - aarnett2
3	Overall project attempted, with sub-projects: Re-writing DumbSAT to use an incremental search through possible solutions
4	Overall success of the project: I believe this project was very successful! I implemented an incremental SAT Solver and re-did the way that DumbSat checked the assignments, which made (marginal) increases in efficiency. I got consistent outputs that aligned with the input test file, leading me to believe the code works even with varying problem sizes and trials. I also plotted a scatter plot that accurately showed the purpose of this project – the sharp increase in computational time requirements as these problems get more complicated. Finally, I learned a ton about how DumbSat works in the first place and SAT problems in general!
5	Approximately total time (in hours) to complete: **~12**
6	Link to github repository: https://github.com/anna-arnett/SATSolver
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	
DumbSat_Incremental_aarnett2.py	Main code file - the Incremental SAT solver, this is the code that runs and generates outputs as well as plots.
Test Files	
data_aarnett2.txt	Provides the necessary problem definitions for the SAT solver. Contributes number of variables, number of clauses, and the clauses themselves. SAT solver

	reads this file solve each test case.
Output Files	
output_aarnett2.txt	This is the result of the SAT Solver. Includes whether each problem is satisfiable or unsatisfiable and records the execution time for solving each problem instance
Plots (as needed)	
timing_plot_aarnett2.png	Provides a visual representation of the performance of the SAT solver across different problem sizes. Specifically, via the plot we can analyze the relationship between the problem size and the execution time taken by the SAT Solver.

8	Programming languages used, and associated libraries: ● Python → code is written in Python ● Libraries ○ Time → used to measure execution time ○ Random → used to build test cases ○ Matplotlib → used to generate the scatter plot
9	Key data structures (for each sub-project): WFF representation was mostly used in a list → there was a list of clauses, and each clause is made up of a list of literals. Literals could either be positive (which represents a variable that is true) or negative (which represents a variable that is false). For example: wff = [[1, -2, 3], [2, 3, -4], [-1, 3, 4]] which represents a formula with three clauses. Lists are also used to track the assignments of True (1) or False (0) for every variable in the SAT problem. Finally, in some parts of the code, test cases are represented as lists or lists of lists. Tuples were also used to store some test case parameters or SAT results in a more fixed format. For example, they were used to store the parameters of a test case including number of variables, number of clauses, and literals per clause.
10	General operation of code (for each subproject):

	The primary task of this code is to solve SAT (Boolean Satisfiability) problems, which involve determining whether there exists an assignment of boolean values (True or False) to a set of variables such that the given boolean formula is satisfied. This code starts by reading a SAT problem from an input file. Each problem consists of a set of variables and a set of clauses (each clause must be satisfied). A clause is represented as a list of literals, where positive integers represent the variable being true and negative integers represent the variable being false. The core of the program is the SAT solver, which checks whether there is a valid assignment of variables that satisfies all the clauses of the boolean formula. Importantly, this code implements an incremental search method, which is a more efficient alternative to the brute-force search method. In a brute-force search, the solver would generate all possible assignments for the variables, then evaluate the entire formula for each assignment. Incremental search improves upon the brute force search by incrementally exploring assignments and immediately stopping as soon as it finds that an assignment is unsatisfiable. Instead of having to generate new assignments, we incrementally change the assignment by using bit-flipping to flip variable values. Using bit-flipping to switch between assignments makes the incremental approach faster compared to brute force, which might generate and test completely new and independent assignments every time. After processing the SAT problem, the code records whether the problem is satisfiable or unsatisfiable and what the execution time was for each problem. The results are written to an output file as well as plotted in a graph.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. I used two primary sets of test cases in the input file to evaluate the performance of the SAT solver. The test cases varied in size, complexity, and the number of variables and clauses which allowed for a relatively broad range of tests. I used a small SAT instance with a problem size of 3 variables and 4 clauses. I believed the small SAT instance was ideal for starting out with testing my program because the number of variables and clauses was relatively small, so it was easy to manually check if the solution was correct or not. This test case told me that the basic logic of the SAT solver was functioning properly, including the ability to generate, check and increment assignments. I also used a medium SAT instance with 8 variables and 16 clauses. Using a larger SAT instance did introduce a little more complexity because of the increased number of variables and clauses. So I was able to test the incremental search in a little more of a realistic scenario. The solver was able to find a solution or determine unsatisfiability within a generally reasonable amount of time, which showed that the incremental search works well for moderately sized problems and that the SAT solver could handle larger sets of clauses and still remain consistent. This was definitely encouraging.
12	How you managed the code development

	Before I started writing code, I made sure that I really understood the SAT Solver and the initial DumbSat code very thoroughly. I paid special attention to the WFF structure and the types of test cases that would be encountered (differing number of variables, literals, etc). When I did start coding, I started by using the DumbSat code and editing the check() function to work on making it incremental. This of course led to a lot of debugging and re-introducing my understanding of what an incremental search was. I used a lot of print statements to debug my code. I also found it relatively easy to use matplotlib for the graphing, as I was able to generate it at the same time as I ran my code. Because I was a single person team, I found it easiest to develop locally on my computer and keep track of my files and code in a VSCode folder. This allowed me to easily run and test my code, and since I didn't have to worry about over-writing a group member's code I think it worked well for me.
13	Detailed discussion of results: I believe the main focus of this project was to recognize how the computational time exponentially increases as the size of the problem (in terms of the number of variables and clauses) increases. The plot provided a strong visual representation of the computational time required for the SAT solver to determine satisfiability or unsatisfiability across problem sizes. The x-axis of the plot represented the number of variables, while the y-axis showed the execution time in microseconds. Green circles represented problems that were found to be satisfiable, while red crosses represented unsatisfiable problems. As expected, the plot showed a sharp and exponential increase in execution time as the problem size increased. For small problems (like problems with 3 or 4 variables), the SAT solver completed the problem very quickly, often in just a few microseconds. The exponential nature of SAT problems became very apparent, particularly after the problem size grew to be above 20 variables. The plot also showed that satisfiable problems often took less time to solve compared to unsatisfiable problems of the same size. This makes sense because once a satisfiable problem is found, the program terminates early without needing to explore all the possible assignments. For example, some satisfiable problems with 8 or 12 variables were solved relatively quickly because the solver found a valid assignment before exploring the entire search space. One key feature that helped curtail the growing computational complexity was the incremental search algorithm which allowed the solver to terminate early when it found a solution for satisfiable problems. The plot visually reflected this optimization because satisfiable cases were clearly often solved in less time. Overall, I think the results effectively showed the key takeaway of the project: computational time increases exponentially with the size of a SAT problem. As the number of variables and clauses grows, the space that the computer needs to search expands exponentially, and even with incremental search optimizations, solving these problems becomes very hard for the computer. The clear exponential trend in the plot highlighted this challenge, which confirmed the theoretical expectation of SAT complexity.

14	How team was organized I was a team of 1 so all of the project duties were mine.
15	What you might do differently if you did the project again If I did this project again, I might want to also try implementing a more advanced algorithm such as the DPLL algorithm and see how that would affect my results with the incremental search. I might be able to create a more powerful solver that would allow me to see more results with less computational time. I would be interested to see what one of those graphs would look like. I could also try implementing a more detailed output visualization such as one that tracks the state of the variable assignments over time or the percentage of the search space used before finding a solution. Either of these things would allow me to dive deeper into the problem, so if I did the project again I might want to try one of these as well.
16	Any additional material: