Project 1 Readme Team highnoonan

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_"team name"

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

<table>
<tr><td>1</td><td>Team Name: highnoonan</td></tr>
<tr><td>2</td><td>Team members names and netids snoonan2</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: Rewrite DumbSAT to use an incremental search through possible solutions</td></tr>
<tr><td>4</td><td>Overall success of the project: My project was overall successful. I was able to rewrite the DumbSAT.py by adding certain techniques, methods, and functions to use and incremental search through all possible solutions. I was successfully able to check that my version of the SAT solver found the same problems satisfiable or unsatisfiable as Professor Kogge's DumbSAT.py. I successfully generated three output files that showed the results of my incremental solver and its systematic way of assigning variables, as well as being clean enough to read and understand by the user. I also successfully created a plot visual to demonstrate the computational complexity of SAT solvers. I displayed the relationship between execution time and problem size, turning out to have an exponential growth curve/rate. Now clearly seeing the complexity and amount of time it takes to solve large instances of wffs, I can see why the need for more efficient SAT solvers is important in any larger program or the computer science field.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 10 hours</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/snoonan2/project1_highnoonan</td></tr>
<tr><td>7</td><td>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.
<table>
<tr><td>File/folder Name</td><td>File Contents and Use</td></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td>code_incramentalSAT_highnoonan.py</td><td>This is my main python program file that rewrites DumbSAT to incrementally go through every possible solution to the wffs. I have just one code file to execute the solving successfully and output all the necessary files. This file contains all the functions to incrementally solve</td></tr>
</table>
</td></tr>
</table>

	the test cases, parse through an input file of test cases if given, write to the output files of the trace, result, and plot. There are also default test cases implemented in the code if no data test case files are included when running the program. This also contains the functions to plot my stored execution time for each problem versus the problem size. It uses libraries such as matplotlib.pyplot and numpy. It also imports time, random, and sys. This code file is based on the DumbSAT.py program file given for us to test and manipulate by Professor Kogge.
Test Files	
data_testcases_highnoonan.txt	**This is a text file that contains my test cases I used on my program for this project. It contains four different cases, with different numbers of clauses and variables, each taking different amounts of time to complete. This test file was used on both my program and the DumbSAT.py from Professor Kogge to ensure the accuracy and success of my program. This holds the test cases that are imputed to create the visualization plot also seen in this repository.**
Output Files	
output_resultsOfRewriteSAT_highnoonan.csv **output_traceOfProgram_highnoonan.csv** **output_cnffile_highnoonan.cnf**	**output_resultsOfRewriteSAT_highnoonan.csv: This output csv file stores the results of running my incremental version of the DumbSAT on the data test cases file in this repository. On each line it contains the problem number, number of variables, number of clauses, number of literals per clause, and total number of literals (clauses * clauses per literal). The line for a given problem also states if the problem was satisfiable or unsatisfiable, a constant value of 1 as a placeholder, the execution time in microseconds, and the values of 0 or 1 for each variable to solve the problem. At the end of the results output file it has the summary of statistics including: name of the CNF file, solver name, total number of problems solved, number of satisfiable problems, number of unsatisfiable problems, total number of problems attempted, and total number of problems completed. This was structured based on the results file found in the DumbSAT.py file given to us by Professor**

	Kogge. output_traceOfProgram_highnoonan.csv: **This output csv file is a trace of my program when run with the test cases found in my data input file. This file also saves what the user would see in their terminal when the program is run. Each line displays an instance of solving a singular problem with information the same as described and saved in the results output file in this repository explained above. After each singular line in the data test case input file, it displays a summary of the number of satisfied and unsatisfied problems and the maximum and average times of the satisfied and unsatisfied problems in a clean manner. I feel that how I have formatted my output trace file makes it clear to identify the characteristics of an individual instance of solving a problem as well as the stats for that solving of overall problems with that size. This helps compare to the DumbSAT.py trace for the correctness of my program and show the user the results in an understandable way.** output_cnffile_highnoonan.cnf: **This output cnf file contains a comment line starting with c that includes: the problem number, the number of literals per clause, and whether the problem is satisfiable (S) or unsatisfiable (U). It also has lines starting with p cnf that specifies the number of variables and clauses. Each line after represents a clause in a cnf formula. The numbers on each line represent the literals in that clause; positive numbers represent positive literals, negative numbers represent negated literals, ending with a 0 to show the end of a clause. This out cnf file provides a standardized way to describe the logical structure of each problem my incremental DumbSAT program generated and attempted to solve.**
Plots (as needed)	
plot_rewriteSATgraph_highnoonan.png	**This png is the plot that is created and saved from my main code program file. It displays that as the size of my problems from my test cases increase, so does the correlating execution time. This visualization shows the success of my understanding the computational complexity of**

	<table><tr><td></td><td>**SAT solving techniques. I have an exponential growth line showing the execution time versus problem size relationship. I also have the satisfiable problems as blue triangles and unsatisfiable problems as red squares to differentiate. For more information on the execution time, I have included lines for both the max execution time and average execution time per problem size.**</td></tr></table>
8	Programming languages used, and associated libraries: **For my program I used Python and python libraries such as, time, random, and sys. I also used the matplotlib.pyplot library to create the plot in my repository and the numpy library to create the exponential growth curve on my plot.**
9	Key data structures (for each sub-project): **I completed one subproject from the SAT solver options with rewriting the DumbSAT.py file to use an incremental search through possible solutions. In my program I used multiple key data structures to successfully complete this project.** **wff (Well-Formed Formula): This is represented as a list of lists in my program. Each inner list represents a clause, and each element in the inner list represents a literal.** **Assignment: This is a list where each index represents a variable, and the value at that index (0 or 1) represents the truth value assigned to that variable. Example: [0, 1, 1, 0, 1] means x1=False, x2=True, x3=True, x4=False, x5=True. This can be seen in my incremental assignment function in the main program.** **TestCases: This is a list of lists seen in my program. It can either be set to my default test cases in the program or from the data input text file (what was used for the output files and plot files in my repository). Each inner list represents a test case with four integers: [Nvars, NClauses, LitsPerClause, Ntrials].** **Results data: These are lists storing various metrics about the solved problems:** **all_timings_sat / all_timings_unsat: the execution times for satisfiable/unsatisfiable problems** **all_problem_sizes_sat / all_problem_sizes_unsat: the problem sizes for satisfiable/unsatisfiable problems** **unique_problem_sizes: the list of unique problem sizes in my program** **max_timings / avg_timings: the maximum and average execution times for each problem size** **These key data structures allow my program to store variable assignments, define**

	test cases, represent boolean formulas, and collect metrics for analysis and visualization.
10	General operation of code (for each subproject) **My code starts with either loading test cases from the data input text file or the default test cases I included. It also sets up the output file names for storing the necessary output information for this project. There is a main solving loop that for each test case the program generates the wffs, attempts to solve using my incremental SAT solver function, and records its satisfiability and execution time. The core of my code is the incremental SAT solver I implemented for the project topic I chose. It tries different variable assignments, going through all the possibilities. It checks for each assignment if all clauses in the formula are satisfied. If an assignment satisfies it, it ends and reports success. If after all possible assignments are made and no satisfiable assignment is made it reports the unsatisfiability. As my program solves the problems, it collects data on the satisfiability results, problem sizes, and execution times. The code then writes the results to my three output files: the results file with the outcomes of each attempt to solve, a cnf file with the boolean formulas, and a trace file with statistics and summaries of the solving similar to the results file. The program then analyzes the collected data to compute the maximum and average solving times to use in my plotting functions. The code finally generates a plot displaying the relationship of execution time versus problem sizes with components on satisfiable and unsatisfiable markers, maximum and average execution time lines, and an exponential growth curve.**
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. **I used both my default test cases that I added directly into my program and the data test cases input text file I created to check the correctness of my code. I used them on both my program and the original DumbSAT.py to ensure that each test case had the same number of satisfiable and unsatisfiable problems, showing that my code could solve the wffs. I then used the same test cases on both my code and the DumbSAT.py to see the plots that were generated for each, seeing that they both held the same sort of shape with the exponential growth relationship of execution time and problem size. This meant implementing my programming functions at the end of the DumbSAT.py file to display the plot which was just a simple addition. The test cases told me that my program could correctly solve the test cases as satisfiable or unsatisfiable using an incremental method, lining up with Professor Kogge's results. It also displayed that my program demonstrated the computational complexity of SAT solvers with the execution time of my test cases having an exponential relationship with their problem sizes.**
12	How you managed the code development **I managed the code development by myself as a solo team, but I felt I managed it well. I first wanted to fully understand the DumbSAT.py file and SAT solvers through the files Professor Kogge gave us before diving right into trying to code a solution. After grasping the brute force method used in DumbSAT.py, I understood I needed to create an incremental assignment function that would systematically assign variables through all the solutions as return or end the loop**

	early on that problem if satisfiability was found - starting with pseudocode. After writing that function, I had to see where in the other functions in DumbSAT.py that needed to be updated to utilize my new function to implement a new way of solving the test cases. The development then turned to how I would display the correctness of my code through a plot, as well as use the structure of the three output files used in DumbSAT.py. I then began to research more on the matplotlib.pyplot library to code the necessary features of my plot to show the relationship between execution time and problem size. These plotting functions were what I worked on last, after checking the correctness of my code with the same data test cases and input text file cases on both mine and Professor Kogge's DumbSAT. There was trial and error with the design of my plot, adding more lines and distinction between unsatisfiable and satisfiable problems.
13	Detailed discussion of results: **This program allowed me to generate results that helped me analyze the performance of the incremental SAT solver across various problem sizes and complexities, providing both numerical data and visual plot representation of the solver's efficiency and scalability. The plot and numerical data stored demonstrated that problem size increases, you should see an overall increase in execution time, having an exponential relationship. The max and average tie lines showed an expected upward trend. The exponential growth line helps visualize how the problem becomes exponentially harder as it grows, understanding through my project the computational complexity of SAT solvers. The results also showed there might be some variation between SAT and UNSAT problems, with one potentially taking longer on average. I thought that due to the incremental solver being more systematic than a brute force solver, it would be faster or more efficient, but that was not always the case. The program also was able to show the results of for each test case, the program generates multiple SAT problems and attempts to solve them (recording if they are satisfiable or unsatisfiable).**
14	How team was organized **Since I completed the project individually, the team was just me. I had the control to organize how I wanted to go about completing the project of rewriting DumbSAT to use an incremental method to go through all the possible combinations. I was able to lay out a framework for myself of understanding the DumbSAT.py file first and SAT solvers as a whole before beginning to code. I then made objectives of what I had to add to the DumbSAT code to achieve the visual display of execution time versus problem size, take input files for test cases, as well as solve the test cases given. I felt that I worked well alone to complete this project successfully. In the future I would like to work in the group to learn how to program in a team dynamic with different coding styles, techniques, and ideas when solving a problem for a theory project. Using Github went smoothly since I was only working on one subproject on one main branch. I also utilized discussing with helpful grad TAs and undergraduate TAs when I had overarching theoretical solving questions, as well as small bugs in my program.**
15	What you might do differently if you did the project again **If I were to do the project again, I would have gone to the grad TAs and undergraduate TAs earlier with my confusions on what the original DumbSAT.py was doing in its program as well as other small questions on my program. As I**

	worked alone, I found myself sometimes getting stuck in my head and not reaching out for help when it was available. I also might have used excel to plot my data instead of using matplotlib.pyplot library, although it might be prettier, because using the library wasn't necessary and it took me a long time. I would also like to try maybe in the future working on a team instead of alone. I think it is important to learn how to code a project or task with a group. This would also help me learn about the communal nature of using GitHub.
16	Any additional material: **n/a**