Project DumbSAT Readme Team mzitella
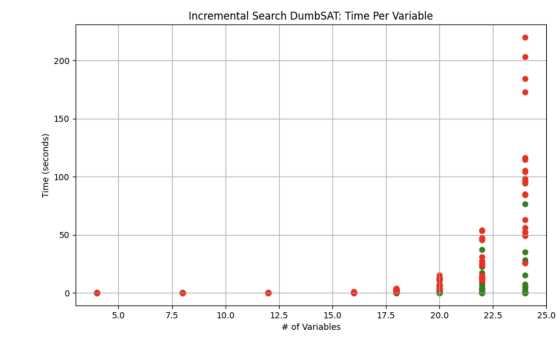
Version 1 9/11/24
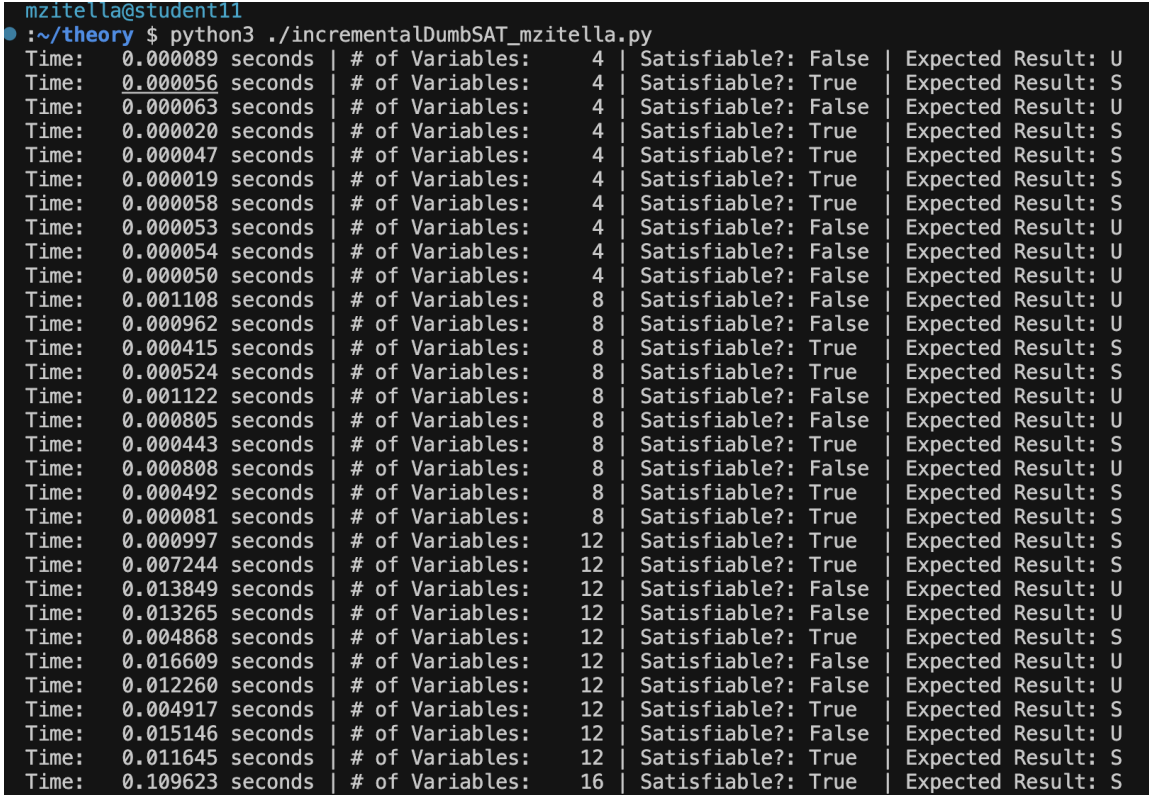
1	Team Name: mzitella
2	Madeline Zitella NetID: mzitella
3	Overall project attempted, with sub-projects: - Rewrite DumbSAT to use an incremental search through possible solutions - Parse the kSAT.cnf.csv data file. Since the file already contains whether the problem is satisfiable or unsatisfiable, I used this information to check the validity of my incremental DumbSAT algorithm - With each SAT problem complete and timed, create a plot using matplotlib, showcasing how the elapsed time increases as the number of variables in a problem increases.
4	Overall success of the project: I created a program that matched the expected satisfiability for each wff in the kSAT file. The kSAT data is very extensive, with several thousand rows of data, which helps prove to me that the incremental search function is indeed successful. In addition, The plot showcases a clear indication that the number of variables greatly impacts how long it takes to determine satisfiability for a problem. This finding reinstills our discussion in class of NP problems, and just how long many-variable problems can take to solve.
5	Approximately total time (in hours) to complete: ~9 hours
6	Link to github repository: https://github.com/MadelineZitella/Theory-Of-Computing-NP-Project
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	
incrementalDumbSAT_mzitella.py	This program builds off of the DumbSAT.py code given, but has an incremental search function, meaning it uses backtracking to find a satisfying variable

	assignment. It takes in the kSAT test data provided and outputs a CSV output file and a plot. To run, simply use: `python3 ./incrementalDumbSAT_mzitella.py` This will generate the output CSV file and the plot. Input data doesn't need to be specified by the user because the kSATfile is already loaded in the program.
Test Files	
kSAT_testdata_mzitella.cnf.csv	This file is on the Canvas page, listed under SAT files. The file contains a long series of different wffs with different numbers of variables, clauses, and literals per clause. 'c' indicates a comment, and 'p' indicates a problem where the clauses follow in the subsequent lines. incrementalDumbSAT_mzitella.py uses this data to check its validity in finding satisfiability
Output Files	
output_mzitella.csv	This file contains the output of incrementalDumbSAT_mzitella.py. From left to right, the columns indicate the number of variables, number of clauses, time in seconds, whether the search found the

		problem was satisfiable or unsatisfiable, and whether the problem was expected to be unsatisfiable or satisfiable, as indicated in the given kSAT data.
	Plots (as needed)	
	plot_mzitella.png	This plot displays the time taken to calculate satisfiability for the number of variables. The red points indicate unsatisfiability and the green points indicate satisfiability.

8	Programming languages used, and associated libraries:
	
	- Python
	    - matplotlib.pyplot as plt was used for plot
	    - Python time library was used to measure the time for each search for the plot and to demonstrate how time increases greatly for difficult problems
	    - Python CSV library was used to parse kSAT data
	    - Python sys library was used to stop the program

9	Key data structures (for each sub-project): The program uses primarily lists and dictionaries to complete each sub-project. For example, in the search_incremental() function, 'assignment' is a list that stores True or False for each variable, first initialized to None before the list is iterated through to attempt different assignments in the backtrack() function. In addition, when processing the kSAT CSV input in process_tests(), clause_arr gets appended and re-initialized as new clauses are read in for each problem. Next, to represent and store multiple SAT problems, I used a list of dictionaries named 'input,' initialized in process_tests(). A dictionary containing information about the number of variables, clauses and satisfiability of a problem is appended to the list of dictionaries each time we have reached the end of the stream of clauses. This organized structure makes it easy to store many SAT problems efficiently and achieve easy access for printing to the terminal.
10	General operation of code (for each subproject) - 1. Rewrite DumbSAT to use an incremental search through possible solutions - This subproject was completed through the program: incrementalDumbSAT_mzitella.py. The functions that handle this subproject are: satisfied(), search_incremental(), and backtrack()

	- The search_incremental() function first calls satisfied() to check the assignment, returning True if at least one literal in the clause is satisfied, and otherwise False. Then, the search function recursively calls backtrack() in order to check every combination of true and false assignments for the problem. If a certain assignment satisfies all of the clauses, then True is returned, indicating satisfiability. If all assignments have been made and the problem is not satisfiable, False is returned. The backtrack function itself tries to assign True first, then False. - 2. Parse the kSAT.cnf.csv data file. Since the file already contains whether the problem is satisfiable or unsatisfiable, I used this information to check the validity of my incremental DumbSAT algorithm - This subproject was completed through the program: incrementalDumbSAT_mzitella.py. The functions that handle this subproject are: process_tests() and run_test(). process_tests() reads in the data from the kSAT CSV file, parsing the comments and problems indicated by 'c' and 'p' respectively. For each SAT problem, a dictionary is stored containing information about variables, clauses and satisfiability. run_test() then calls process_tests() to parse the CSV file before calling measure_time() which calls and times how long the search_incremental() takes. run_test() also outputs a CSV that contains the number of variables, number of clauses, elapsed time, the search_incremental() function's determination of satisfiability, and the expected satisfiability (U or S). The function also prints out results to the terminal (see screenshot at the end of the page). - 3. With each SAT problem complete and timed, create a plot using matplotlib, showcasing how the elapsed time increases as the number of variables in a problem increases. - This subproject was completed through the program: incrementalDumbSAT_mzitella.py. The functions that handle this subproject are: measure_time() and plot_results(). measure_time() computes the elapsed time it takes to run search_incremental on a problem which is later used for the plot's y-axis. plot_results() uses matplotlib to create a graph showcasing how the time it takes to check satisfiability for a problem increases at a rapid rate as the number of variables in the problem increase. I used many matplotlib tools to color the data points, add labels, a title, a legend, and other features to create a clear model.

11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code.
	
	To check that my search_incremental() function was correctly determining whether a problem was unsatisfiable or satisfiable, I utilized the kSAT.cnf.csv dataset. in the file, it is indicated whether the problem is satisfiable (S) or unsatisfiable (U). This information allowed me to easily check whether my algorithm was correct, again known solutions. I performed this check in the run_test() function:
	`  if (my_satisfy and satisfiability == 'U') or (not my_satisfy and`
	`satisfiability == 'S'):`

| | ```
            # if results do not match we exit the program
            sys.exit("Stopping.")
``` |
| | 'my_satisfy' indicates satisfiability that I derived from the incremental_search() function, and 'satisfiability' is taken from the kSAT CSV file. This check immediately exits the program if my own program does not match that of the known result. While a very simple check, this code allowed me to completely verify that my program works as expected. |
| 12 | How you managed the code development

I began with thoroughly reviewing the DumbSAT.py code provided and noted what changed needed to be made to implement an incremental search. I realized early that the test data I chose would dictate how my program would parse and process the CSV file. I chose the kSAT.cnf.csv data set because I knew having the satisfiability answers for each problem already indicated would make verifying my program simple. After choosing the data, I started by creating the incremental search function, first writing mostly in pseudocode and then gradually implementing code and running it with print lines to debug. I then focused on how to process the CSV data and focused on staying organized. I researched file handling in order to open the input file and save an output CSV file as well. Finally, I turned to plotting the data using matplotlib. While I first had the code to plot the data in a separate file, I ultimately consolidated all functions into one file for ease. Throughout the entire process I started with pseudocode, testing with print lines frequently and thought often about changing my approach or thinking about the problem differently, especially when it came to developing the incremental search and backtracking functions. |
| 13 | Detailed discussion of results:

The results of this project are most clearly indicated on the above plot, which shows how the elapsed time to solve each problem increased dramatically as the number of variables in the problem increased. In the worst case, a problem with just under 25 variables took over 3 minutes to check all assignments and determine satisfiability. It is also evident that there is a bottleneck in the data; as the number of variables reach more than twenty, there is a sharp upturn in elapsed time. This bottleneck demonstrates how time-intensive it is to solve SAT problems due to the sheer number of assignments to check. In addition, it makes sense that unsatisfiable problems (indicated in red) would take more time to solve than satisfiable problems (indicated in green) since all assignments are searched through before determining unsatisfiability. This concept agrees with the graph, as most of the green data points fall lower than the red. The |

	results as a whole demonstrate the difficulty with a brute-force incremental approach, and show that some kind of heuristic or smart-decision making would greatly reduce the number of assignments to check, and therefore elapsed time.
14	How team was organized

N/A, worked individually |
| 15 | What you might do differently if you did the project again

If I did this project again, I would be interested in obtaining several datasets with different ranges of variables, literals, clauses, etc in order to generate several graphs to examine different models and trends of how the amount of variables can affect the elapsed time, instead of simply one graph. In addition, I would be interested in taking the same test data I worked with and modifying my search algorithm to contain some sort of heuristic to measure how much the elapsed time improves for each problem. Although that would turn into a different project entirely, I would be very interested to see how a smarter search algorithm could affect the time and if the same bottleneck would occur. |
| 16 | Any additional material:

Screenshot of terminal after running python3 ./incrementalDumbSAT_mzitella.py to prove that program runs |

```
mzitella@student11
:~/theory $ python3 ./incrementalDumbSAT_mzitella.py
Time:    0.000089 seconds | # of Variables:    4 | Satisfiable?: False | Expected Result: U
Time:    0.000056 seconds | # of Variables:    4 | Satisfiable?: True  | Expected Result: S
Time:    0.000063 seconds | # of Variables:    4 | Satisfiable?: False | Expected Result: U
Time:    0.000020 seconds | # of Variables:    4 | Satisfiable?: True  | Expected Result: S
Time:    0.000047 seconds | # of Variables:    4 | Satisfiable?: True  | Expected Result: S
Time:    0.000019 seconds | # of Variables:    4 | Satisfiable?: True  | Expected Result: S
Time:    0.000058 seconds | # of Variables:    4 | Satisfiable?: True  | Expected Result: S
Time:    0.000053 seconds | # of Variables:    4 | Satisfiable?: False | Expected Result: U
Time:    0.000054 seconds | # of Variables:    4 | Satisfiable?: False | Expected Result: U
Time:    0.000050 seconds | # of Variables:    4 | Satisfiable?: False | Expected Result: U
Time:    0.001108 seconds | # of Variables:    8 | Satisfiable?: False | Expected Result: U
Time:    0.000962 seconds | # of Variables:    8 | Satisfiable?: False | Expected Result: U
Time:    0.000415 seconds | # of Variables:    8 | Satisfiable?: True  | Expected Result: S
Time:    0.000524 seconds | # of Variables:    8 | Satisfiable?: True  | Expected Result: S
Time:    0.001122 seconds | # of Variables:    8 | Satisfiable?: False | Expected Result: U
Time:    0.000805 seconds | # of Variables:    8 | Satisfiable?: False | Expected Result: U
Time:    0.000443 seconds | # of Variables:    8 | Satisfiable?: True  | Expected Result: S
Time:    0.000808 seconds | # of Variables:    8 | Satisfiable?: False | Expected Result: U
Time:    0.000492 seconds | # of Variables:    8 | Satisfiable?: True  | Expected Result: S
Time:    0.000081 seconds | # of Variables:    8 | Satisfiable?: True  | Expected Result: S
Time:    0.000997 seconds | # of Variables:   12 | Satisfiable?: True  | Expected Result: S
Time:    0.007244 seconds | # of Variables:   12 | Satisfiable?: True  | Expected Result: S
Time:    0.013849 seconds | # of Variables:   12 | Satisfiable?: False | Expected Result: U
Time:    0.013265 seconds | # of Variables:   12 | Satisfiable?: False | Expected Result: U
Time:    0.004868 seconds | # of Variables:   12 | Satisfiable?: True  | Expected Result: S
Time:    0.016609 seconds | # of Variables:   12 | Satisfiable?: False | Expected Result: U
Time:    0.012260 seconds | # of Variables:   12 | Satisfiable?: False | Expected Result: U
Time:    0.004917 seconds | # of Variables:   12 | Satisfiable?: True  | Expected Result: S
Time:    0.015146 seconds | # of Variables:   12 | Satisfiable?: False | Expected Result: U
Time:    0.011645 seconds | # of Variables:   12 | Satisfiable?: True  | Expected Result: S
Time:    0.109623 seconds | # of Variables:   16 | Satisfiable?: True  | Expected Result: S
                          Ln 149, Col 12    Spaces: 4   UTF-8   LF   { } Python   3.12.4 64-bit
```