Project Readme Template

16/11/25

1	Team Name: Dishy Wizard
2	Team members names and netids: Jack Frauenhofer (jfrauenh)
3	Overall project attempted, with sub-projects: Brute-Force SAT Solver
4	Overall success of the project: Successful
5	Approximately total time (in hours) to complete: 12
6	Link to github repository: https://github.com/jackFrauenhofer/Project1-TOC
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)

File/folder Name	File Contents and Use
Code Files	
plot_results-DishyWizard.py	Used to plot the results from the brute force SAT solver
sat.py	Updated the ‘sat_bruteforce’ function.
reformatcnf-DishyWizard.py	
Test Files	
cnffile.cnf	Basic CNF file input to test Brute Force SAT solver on few instances. This file was included in the original github clone as a basic test. I used this file to mainly test that the output of my solver was working before running the CNF’s with much more instances.
check-kSAT.cnf-Dishy Wizard	CNF file that contains instances of various k-complexities to test Brute Force SAT solver. This file was sourced from the canvas resource page.
check-2SAT.cnf-Dishy Wizard	CNF file that contains instances of only k=2 complexity SATs to test Brute Force SAT solver. This file was sourced from the canvas resource page.
Output Files	

<table>
<tr><td></td><td>
<table>
<tr><td>brute_force_cnffile_sat_solver_results-DishyWizard.csv</td><td>CSV file that includes the output from running Brute Force on input file ‘cnffile.cnf’</td></tr>
<tr><td>brute_force_kSAT_sat_solver_results-DishyWizard.csv</td><td>CSV file that includes the output from running Brute Force on input file ‘check-kSAT.cnf-DishyWizard’.</td></tr>
<tr><td>brute_force_2SAT_sat_solver_results-DishyWizard.csv</td><td>CSV file that includes the output from running Brute Force on input file ‘check-kSAT.cnf-DishyWizard’.</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td>plots-brute_force_cnffile-DishyWizard.png</td><td>Simple plot analyzing various execution times compared to different instances. This plot is not materially helpful in analyzing different complexities and their time executions - it was more to test if the output of my ‘plot_results.py’ was working.</td></tr>
<tr><td>plots-brute_force_kSAT-DishyWizard.png</td><td>Plot analyzing various execution times compared to different instances with various k-complexities</td></tr>
<tr><td>plots-brute_force_2SAT-DishyWizard.png</td><td>Plot analyzing various execution times compared to different instances, all with k=2 complexities.</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries:
- Python
Libraries:
- csv
- Itertools
- Numpy
- Matplotlib.pyplot
- Json
- Os
- time</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):

For the Brute Force SAT solver function, the key data structures were:

- In the function ‘generate_solutions’, I use a generator named ‘assignment’ to yield new assignments when needed, rather than loading all assignments at once. The assignment generated is a list, where each index correlates to a different variable, and the list is boolean.
- I use a dict named ‘solution’ to map any successful assignment to the proper output

For the plot_results-DishyWizard.py script:</td></tr>
</table>

	- x, y, and colors are all lists. These lists are to be used to make the x-axis, the y-axis, and color of each marker respectively. They are made by parsing the output csv file's columns
10	General operation of code (for each subproject): For the Brute Force SAT Solver: - I built two helper functions to help run this solver. 'Generate_solutions' is a generator that yields a new assignment when called in a for loop. This allowed me to look at one assignment instead of loading all of them before, thus reducing the best case time complexity of the algorithm. For each assignment that was generated, 'satisfiable_verifier' would check if the assignment is valid by seeing if each clause in the SAT was true. If all clauses evaluated to true, the assignment would be deemed successful, and the SAT said to be satisfiable. If the assignment failed just one of the clauses, a new assignment would be generated and verified again, until either an assignment is successful, or no assignment is, in which case the SAT would be deemed unsatisfiable.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. - I used the two CNF files from canvas (ksat.cnf and 2sat.cnf) in order to verify the correctness of my code and its output. These files were also helpful for me to analyze the exponential increase in time complexity these SAT problems have.
12	How you managed the code development - I built the program by meticulously checking different sections and verifying that they are giving correct outputs. I did this using many print() functions. At first, I had built the brute force algorithm without a generator. Instead, I used a list comprehension to create every assignment at once, but I quickly realized this was inefficient because we could check an assignment every time it is generated, which could save time in the best case. Thus, I implemented a generator to yield assignments one at a time. This also would allow concurrent processing to speed up the algorithm. - After I checked that I was receiving the correct output for each of the three cnf files that I used as testing, I built the plot_results-DishyWizard.py' script to plot the csv outputs into plots that I stored in the 'figures' folder.
13	Detailed discussion of results: Based on the results of the two valuable tests I did on the cnf files named 'ksat.cnf' and '2sat.cnf', I analyzed that, in worst-case scenarios (when SAT is unsatisfiable), the algorithm follows an exponential trend as the number of variables increases. The worst-case scenario time complexity could thus be considered as O(2^n). When the SAT was satisfiable, the time complexity still increased on average at approximately O(2^n), but the best-case scenario could be O(1) if the first assignment

	generated was successful at satisfying the SAT. If the SAT was satisfiable, the average time complexity seemed to be half of the worst-case, but in reality, it is completely random as to when the successful assignment would be yielded by the generator. In conclusion, the brute force algorithm has an exponential time complexity on average.
14	How team was organized: I was a solo team.
15	What you might do differently if you did the project again: - In general, the program could be improved by implementing the 'plot_results.py' script to be run after each input when 'uv run main.py' is run. In the github, we were warned not to interfere with this script, so I thought it was best to manually create the plots. However, I think automating this process of updating the input_file for the plot into the direct function when the output CSV file is created would help automate the process. Thus, I would improve the automation of the program to eliminate the need to manually change the input / output file names. - Although I signed up to build the 'sat_bruteforce' function, there are clearly better, more time-effective ways to solve SAT problems. Brute force is a relatively basic, less time-efficient solver than backtracking or other solvers. Thus, I would want to improve my solver by implementing the other more effective solvers. - To improve the time length of the Brute Force SAT solver, I would implement concurrent processing. I have already implemented a generator, so changing the function to run with the 6 multicores I have on my mac would tremendously advance and speed up the program's run time.
16	Any additional material: None