Project TeamworkTemplate

Version 1 9/11/24

A ***separate copy*** of this template should be filled out and submitted by each student, regardless of the number of students on the team. Also change the title of this template to “Project x Teamwork <team> - <netid>”

<table>
<tr><td>1</td><td>Team Name: Trigonometry</td></tr>
<tr><td>2</td><td>Individual name: Ella Trigiani</td></tr>
<tr><td>3</td><td>Individual netid: etrigian</td></tr>
<tr><td>4</td><td>Other team members names and netids: N/A</td></tr>
<tr><td>5</td><td>Link to github repository: https://github.com/ellatrigiani/Project1-TOC.git</td></tr>
<tr><td>6</td><td>Overall project attempted, with sub-projects: SAT Bruteforce</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)
<table>
<tr><th>File/folder Name</th><th>File Contents and Use</th></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td>sat.py/src</td><td>

Class Definition (SatSolver) was given to us and provides infrastructure for file I/o, CSV writing, and plotting
Brute force solver (sat_bruteforce)

Takes n_vars, which is the number of boolean variables, and clauses as input
It returns tuple saying if the CNF is satisfiable and what the solution is
It performs edge case checks, just in case the test input calls for them

These return true or false if any clause is empty

Creates an assignment dictionary mapping variables numbers (0s and 1s) to true and false
Calls recursive helper function (incremental_sat)
Converts internal format (0s and 1s) to output format (True and False) using dictionary comprehension

Checking Clause Satisfaction (is_clause_satisfied)

For each literal in a clause, check if it is

</td></tr>
</table>
</td></tr>
</table>

	satisfies the current assignment ○ Positive literals are satisfied when the variable is true ○ Negative literals are satisfied when the variable is false ○ returns true if at least one literal in the clause is satisfied ● Checking formula satisfaction (formula_satisfied) ○ Iterates through all clauses in the formula ○ Returns false immediately if any clause is unsatisfied ○ Returns true only if all clauses are satisfied ● Incremental SAT solver (incremental_sat) ○ Recursive DFS through all possible assignments ○ Base case: when depth > n_vars, all variables are assigned, check if the formula is satisfied ○ Recursive case: trying to assign current variable to true then recursively solve for remaining variables ○ If true doesnt work go back and try false ○ If neither works remove the assignment and return false ○ Key characteristic of brute force: only checks formula satisfaction when all variables are assigned, rather than on partial assignments ○ Explores all $O(2^n)$ assignments
sat_runner_trigonometry.py/scripts	● Path setup ○ Needed this root to the python path to enable imports from src/, otherwise it was failing ○ Allows script to be run from anywhere while accessing helper functions ● Importing helper functions ○ Needed to parse CNF files ○ Import SatSolver class to call the brute force solving method ○ Imports matplotlib and numpy so I could make the graphs ● Writing results to a CSV (write_results_csv) ○ Creates output directory if it DNE ○ Opens CSV file and writes the proper header ○ For each parsed instance we see the time of the execution, if it is satisfiable, and the solution

	○ Collects timing data and variable counts for plotting ○ Returns lists needed to make graph ● Generating Plot (plot_from_results) ○ Creates scatter plot with number of variables (x_axis) vs execution time in seconds (y_axis) ○ Separates data points by satisfiability ○ Plots worst case time complexity ○ Demonstrates exponential growth increase ○ Saves as a png file ● Main ○ Parses command line arguments ○ Uses provided parser to extract instances from input file ○ Creates SatSolver instance ○ Calls CSV writing function to solve all instances and record results ○ Generates performance plot in same dictionary as CSV output ○ Provides command-line interface: uv run scripts/sat_runner_trigonometry.py input/data_trigonometry.cnf results/output_trigonometry.csv
Test Files	
cnffile.csv/input	● Test file provided to us in CNF format
data_trigonometry.csv/input	● Smaller version of the 2SAT file given to us, still over 1000 lines, but easier to run in an efficient amount of time to see the output
Output Files	
brute_force_cnffile_sat_solver_results.csv/results	● Output file produced from the test input provided to us ● Automatically created when we run uv run main.py ● Stored in the results directory ● Outputs in a CSV file format with the correct columns we were given
output_trigonometry/results	● Same gist as the previous output file, but this is run on a separate input I put in ○ I got the input from the 2SAT file on canvas but only took the first ~1000 lines because I wanted it to be large, but not take forever to run ● Outputs a CSV that highlights satisfiability in the

<table>
<tr><td></td><td>
<table>
<tr><td></td><td>same order as the previous output
● Ran on a lot more CNFs so much larger than the previous file</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td>output_graph_trigonometry.png/results</td><td>● Generated from plot_from_results in the sat_runner_trigonometry.py
● Generated after solving for all instances and writing the CSV outputs
● Saved as a PNG image in the same directory as CSV output
● Graph components
○ X-axis is the number of variables which shows how complexity increases with more variables
○ Y-axis is the time in seconds, which shows the actual runtime of the brute force algorithm
○ The green circles are the satisfiable problems
○ The triangles are the unsatisfiable problems
○ The dotted blue line is the theoretical complexity of 2^n, which we can see the points sort of follow the line
● This graph very clearly displays that as the number of variables increase the runtime grows exponentially</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Individual Student time (in hours) to complete: 10 hours</td></tr>
<tr><td>9</td><td>Your specific activities and responsibilities: Building the entire program, so filling in the given SAT class code as well as writing helper functions. I also built a program to run the code locally with my own input, which outputs a results csv file and a graph png.</td></tr>
<tr><td>10</td><td>What was personally learned (topic, programming, algorithms):

Through this project, I deepened my understanding of the SAT problem and its classification. I learned how to implement a brute force SAT solver using bruteforce, which explores all possible truth assignments (2^n combinations) to determine if a CNF formula is satisfiable.

On the programming side, I definitely felt challenged since this was such a dynamic program with many moving pieces. At the end of the day, my Python skills definitely improved and I’ve become more comfortable working with many different data structures.

The most valuable algorithmic insight was understanding the differences between brute</td></tr>
</table>

	force and backtracking, since my approach is brute force but still required some recursion. My implementation uses a recursive DFS that tries both True and False for each variable, which demonstrates the exponential $O(2^n)$ time complexity characteristic. I think this project also provided some real world software engineering skills, since we had to balance many module imports, working with classes, using uv, and organizing code given certain templates.
11	How the team was organized, and what might be improved: If I were to approach this differently, if I could do it over again, I would: 1. Start earlier, which would give me more time to understand the infrastructure before implementing the algorithm. 2. Read all documentation thoroughly first, to understand the file structure and the use of main.py.Some of the instructions and I didn't exactly know what to do, so taking a more thorough approach would have helped clarify the confusion. 3. Separate concerns better, like creating a standalone test script early on to verify my algorithm was working. I did this more towards the end, but it probably would've been smarter to do it in the beginning. 4. Lastly, definitely ask for clarification sooner. Some aspects of the file format, what we were supposed to produce, and expected output weren't immediately clear, but other than that it was good.
12	Any additional material: Not really. I was just going to add that I think it would be better for the instructions to be a bit clearer. There are a lot of moving parts and it can get a little messy to understand. But overall, I do think this helped me learn a lot since it was complicated: both about SAT solving and general Python usage. Also just wanted to point out again, in my own personal runner the reason I wrote a csv parsing file was because calling the helper function given to us was causing a lot of problems on my end. I also didn't want the file to be named what the parser the TA assigned names the output file, but couldn't edit that because it is needed for the autograder. It also was pretty simple to implement.