Project Readme Template

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_"teamname"

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

<table>
<tr><td>1</td><td>Team Name: cander35</td></tr>
<tr><td>2</td><td>Team members names and netids: cander35</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: I think I was successful!</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 20 hours</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/cateranderson/TOC_Project1.git</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><th>File/folder Name</th><th>File Contents and Use</th></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td>DumbSATIncrementer_parseCSV_plotData_cander35.py</td><td>- Parsing CNF Problems (parse_cnf_csv):
- The program reads a CSV input file (or stdin) containing multiple CNF problems.
- It skips comments and identifies the CNF format via the p cnf line, extracting the number of variables (Nvars) and clauses (Nclauses).
- Clauses (sets of literals) are stored in lists, and each problem is appended to the problems list as a tuple: (wff, Nvars, Nclauses).</td></tr>
</table>
</td></tr>
</table>

	- Checking Clause Satisfaction (is_clause_satisfied):
	  - For each clause, the program checks if at least one literal satisfies the clause.
	  - For positive literals, if the corresponding variable is assigned True (1), the clause is satisfied.
	  - For negative literals, if the corresponding variable is assigned False (0), the clause is satisfied.
	- Incremental SAT Solver (incremental_sat):
	  - A recursive depth-first search tries to assign truth values to variables.
	  - For each variable, it first assigns True (1) and recursively checks if this leads to a satisfying assignment for all clauses.
	  - If not, it backtracks and tries assigning False (0).
	  - The base case is reached when all variables are assigned, and it checks if all clauses are satisfied.
	- Testing a WFF (test_wff):
	  - For each problem (WFF), the solver tests whether a satisfying assignment exists.
	  - It records the execution time taken to either find a satisfying assignment or determine the problem is unsatisfiable.

Test Files	
kSAT_input_cander35.cnf.csv Originally Dr. Kogge's kSAT.cnf.csv	- Parsing the CSV properly for useful input (parse_cnf_csv) - The CSV file contains CNF problems with comment lines

<table>
<tr><td></td><td>
<table>
<tr><td>file, but name edited</td><td>starting with `c`, problem definition lines starting with `p` (specifying the number of variables and clauses), and each clause represented by a list of integers (positive for variables, negative for their negations). Clauses end with `0` or may omit it, and empty lines or extra spaces are ignored.</td></tr>
<tr><td colspan="2">Output Files</td></tr>
<tr><td>output_cander35.txt</td><td>- Logging Results (log_results) into this .txt file:
- The results for each problem (problem number, number of variables, satisfiability, execution time, and the satisfying assignment if applicable) are written to an output file or standard output.</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td>sat_solver_performance_cander35.png</td><td>- Plotting the Results (plot_data) into this .png file:
- After solving the problems, the program reads the output file and plots the number of variables against the execution time.
- Satisfiable problems are marked with green circles, and unsatisfiable ones with red triangles.
- A line of best fit representing the worst-case complexity (2^n) is also plotted.</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries: Python- time, csv, sys, matplotlib.pyplot, numpy</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):</td></tr>
</table>

	- List:
	  - Problems list: Stores the parsed CNF (Conjunctive Normal Form) problems from the input. Each problem consists of a tuple containing the WFF (well-formed formula), the number of variables (Nvars), and the number of clauses (Nclauses).
	  - WFF list: Stores individual clauses, where each clause is itself a list of literals (integers representing variables or their negations).
	  - Clauses list: Inside WFF, each clause is represented as a list of integers (positive integers for variables and negative integers for negated variables).
	- Dictionary:
	  - Assignment dictionary: Holds the current truth assignment for each variable. The key is the variable number, and the value is 1 (True) or 0 (False).

10	General operation of code (for each subproject):

11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code.:
	- Parsing CNF Problems (parse_cnf_csv):
	  - The program reads a CSV input file (or stdin) containing multiple CNF problems.
	  - It skips comments and identifies the CNF format via the p cnf line, extracting the number of variables (Nvars) and clauses (Nclauses).
	  - Clauses (sets of literals) are stored in lists, and each problem is appended to the problems list as a tuple: (wff, Nvars, Nclauses).
	- Checking Clause Satisfaction (is_clause_satisfied):
	  - For each clause, the program checks if at least one literal satisfies the clause.
	  - For positive literals, if the corresponding variable is assigned True (1), the clause is satisfied.
	  - For negative literals, if the corresponding variable is assigned False (0), the clause is satisfied.
	- Incremental SAT Solver (incremental_sat):
	  - A recursive depth-first search tries to assign truth values to variables.
	  - For each variable, it first assigns True (1) and recursively checks if this leads to a satisfying assignment for all clauses.
	  - If not, it backtracks and tries assigning False (0).
	  - The base case is reached when all variables are assigned, and it checks if all clauses are satisfied.
	- Testing a WFF (test_wff):
	  - For each problem (WFF), the solver tests whether a satisfying assignment exists.

	- It records the execution time taken to either find a satisfying assignment or determine the problem is unsatisfiable.
	- Logging Results (log_results):
	  - The results for each problem (problem number, number of variables, satisfiability, execution time, and the satisfying assignment if applicable) are written to an output file or standard output.
	- Plotting the Results (plot_data):
	  - After solving the problems, the program reads the output file and plots the number of variables against the execution time.
	  - Satisfiable problems are marked with green circles, and unsatisfiable ones with red triangles.
	  - A line of best fit representing the worst-case complexity (2^n) is also plotted.

12	How you managed the code development: To tackle this incremental search approach, I focused on several key steps to enhance efficiency. Instead of generating all possible assignments upfront, I decided to build partial assignments clause-by-clause. If a partial assignment failed to satisfy a clause, I backtracked immediately, avoiding irrelevant combinations. I utilized a recursive function that explores potential solutions in a depth-first manner, setting one variable at a time and checking its consistency with the constraints. If a partial assignment was valid, I proceeded; if not, I backtracked. Key changes in my approach included replacing the complete enumeration of assignments with this recursive function, which tests each variable in both possible states (True or False) incrementally. I also implemented an early exit strategy, stopping the search once a solution is found, which reduced the number of assignments checked. Lastly, I ensured that if any partial assignment failed to satisfy a clause, I backtracked right away, further streamlining the process. Additionally, I started testing with smaller self made examples that I knew the solutions to. Once I knew this worked, I started the csv process and had code that parsed simple csv files I made. Then once this worked, I started importing Dr. Kogge's large test files. Once I got this to work and had my output formatted the way I thought was most useful and readable I started working on parsing the output to create a useful plot that could inform users on the output of large csv files with many different variables and satisfaction outcomes.
13	Detailed discussion of results: The approximate complexity of my program was O(2^n). This is because the overall complexity is O(P * 2^Nvars * Nclauses), where P is the number of problems, Nvars is the number of variables, and Nclauses is the number of clauses per problem. The exponential complexity arises because the algorithm checks every possible truth assignment (which grows exponentially with the number of variables) in order to find a satisfying assignment.

	My output generally demonstrated that the greater the number of variables, the longer the computation times were. Additionally, the greater the number of variables, the time it took to reach a result of unsatisfactory took much longer than with a smaller number of variables. This highlights the exponential growth in difficulty as the number of variables increases. I feel that the plotted data demonstrates this pattern quite well. The plot illustrates the performance of the SAT solver by depicting the relationship between the number of variables and the execution time in microseconds. On the x-axis, the number of variables is plotted, while the y-axis shows the corresponding execution time. Two types of problems are represented: green circles indicate satisfiable problems, and red triangles denote unsatisfiable ones. Additionally, a dashed blue line highlights the theoretical worst-case time complexity, which follows an exponential O(2^n) growth pattern, illustrating the expected performance for unsatisfiable problems. This plot provides a clear visual comparison between the actual performance of the solver and the theoretical limits, demonstrating how the execution time scales as the problem size increases. You can see that as the number of variables increases, the amount of red triangles depicted also increases at a higher y-axis location. To tackle this incremental search approach, I focused on several key steps to enhance efficiency. Instead of generating all possible assignments upfront, I decided to build partial assignments clause-by-clause. If a partial assignment failed to satisfy a clause, I backtracked immediately, avoiding irrelevant combinations. I utilized a recursive function that explores potential solutions in a depth-first manner, setting one variable at a time and checking its consistency with the constraints. If a partial assignment was valid, I proceeded; if not, I backtracked. Key changes in my approach included replacing the complete enumeration of assignments with this recursive function, which tests each variable in both possible states (True or False) incrementally. I also implemented an early exit strategy, stopping the search once a solution is found, which reduced the number of assignments checked. Lastly, I ensured that if any partial assignment failed to satisfy a clause, I backtracked right away, further streamlining the process. Additionally, I started testing with smaller self made examples that I knew the solutions to. Once I knew this worked, I started the csv process and had code that parsed simple csv files I made. Then once this worked, I started importing Dr. Kogge's large test files. Once I got this to work and had my output formatted the way I thought was most useful and readable I started working on parsing the output to create a useful plot that could inform users on the output of large csv files with many different variables and satisfaction outcomes.
14	How the team was organized : I worked alone.
15	What you might do differently if you did the project again: I think what I need to improve is the process of developing my code. I feel that I have a lot of different functionality in just one script, but I think the next time I do a large project like this one, different

	components should be handled in different scripts for better organization and for debugging purposes. For example a script to parse the input and clean it up, and then a separate script to analyze and test the WFF and produce output, and then another script to plot the output data might have made the process smoother.
16	Any additional material: n/a