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"

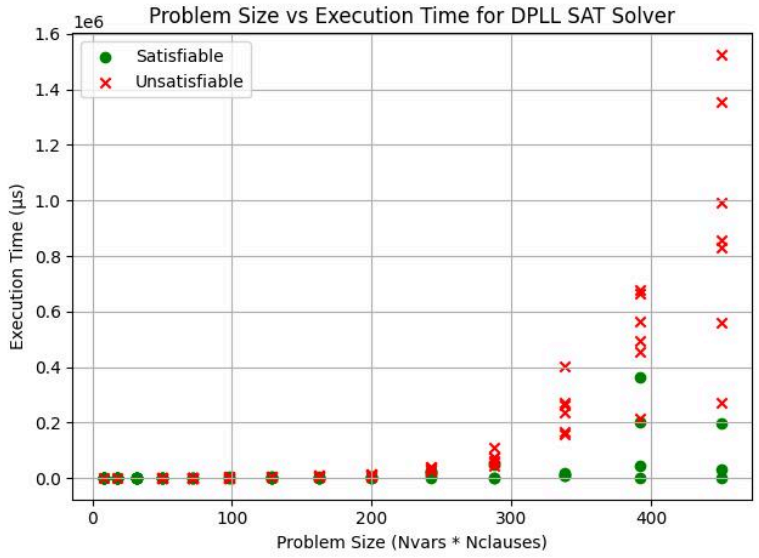
Also change the title of this template to "Project x Readme Team xxx"

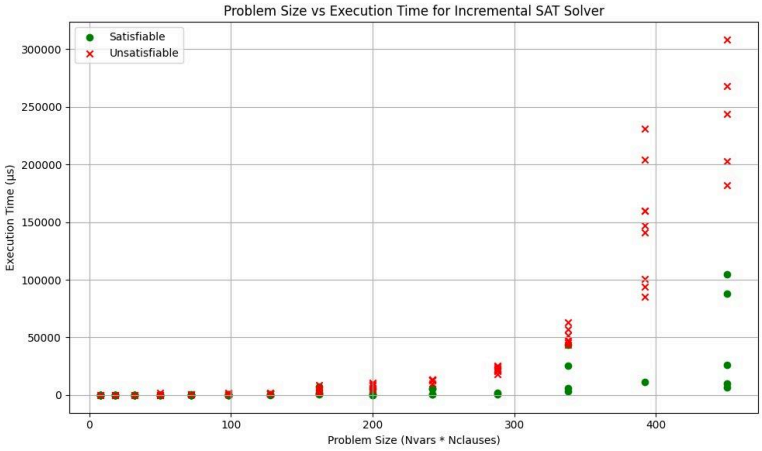
<table>
<tr><td>1</td><td>Team Name: Isabel_Phoebe_Pablo</td></tr>
<tr><td>2</td><td>Team members names and netids: Phoebe Huang (chuang26), Pablo Oliva Quintana (polivaqu), Isabel Ojeda (iojeda)</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: DumbSAT

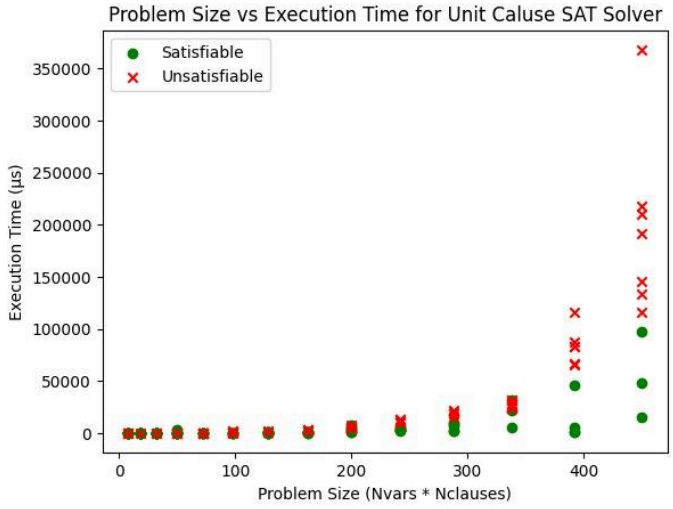
1. Implementing a polynomial time 2-SAT solver (look up the DPLL algorithm)
2. Rewrite DumbSAT to use an incremental search through possible solutions
3. Rewrite DumbSAT to use "unit clause" rules as much as possible
4. Rewrite DumbSAT to do both</td></tr>
<tr><td>4</td><td>Overall success of the project: Successful in our opinion</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: (3-4 hours per person) → 12 hours in general per day for 5 days.</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/iojeda1/TheoryProject1.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><td>File/folder Name</td><td>File Contents and Use</td></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td>1. DumbSAT_DPLL_Final.py
2. DumbSAT_IS.py
3. DumbSAT_UnitCaluse.py</td><td>1. Implementing DPLL algorithm to find if certain wff formulas are satisfiable or unsatisfiable.
2. Implementing incremental search algorithm to find if certain</td></tr>
</table>
</td></tr>
</table>

	wff formulas are satisfiable or unsatisfiable. 3. Implementing unit clause algorithm to find if certain wff formulas are satisfiable or unsatisfiable.
Test Files	
Included in each code file as functions.	[[1, -2], [-1, 2]] [[1, 2], [-1, 2], [1, -2]] [[1, 2, 3], [-1, -2], [2, -3], [1, 2]] [[1, 2], [1, -2], [2, 3], [-1, 3], [-3, -2]] [[1], [-1]]
Output Files	
1. dpll_results.csv 2. incremental_results.csv 3. unit_results.csv	Output includes the wff formula, whether is is satisfiable or not, the satisfiable assignment if it is satisfiable, and the execution time for individual and total test cases.
Plots (as needed)	


Problem Size vs Execution Time for DPLL SAT Solver
1e6
Satisfiable
Unsatisfiable
Execution Time (µs)
Problem Size (Nvars * Nclauses)


Problem Size vs Execution Time for Incremental SAT Solver
Satisfiable
Unsatisfiable
Execution Time (µs)
Problem Size (Nvars * Nclauses)


Problem Size vs Execution Time for Unit Caluse SAT Solver
Satisfiable
Unsatisfiable
Execution Time (µs)
Problem Size (Nvars * Nclauses)

8	Programming languages used, and associated libraries: Python (libraries = time, random, string, csv)
9	Key data structures (for each sub-project): 1. DPLL = Sets and lists 2. Incremental Search = Lists 3. Unit Clause = Lists
10	General operation of code (for each subproject) 1. Incremental search: a. The code begins with an initial truth assignment where each variable is randomly assigned either true or false. The algorithm checks if the current assignment satisfies all clauses in the wff formula. It iterates through each clause and checks if at least one clause in the literal evaluates to True. If the formula is not satisfied, the algorithm randomly flips the value of a single variable. This repeats for a max 2^Nvars number of attempts and ends when a satisfying assignment is found or after reaching the maximum number of attempts. 2. Unit clause propagation: a. The unit clause function searches for unit clauses in the formula. When a unit clause is found (clause with one literal), it forces the truth value of that literal (if it is positive, the variable is set to True; if it is negative, it is set to False). Once a literal truth value is determined by a unit clause, the formula is simplified by removing clauses containing the literal are removed (as they are satisfied) and clauses containing the negation of the literal have that negated literal removed (since it is falsified). The process repeats until no more unit clauses are found. After simplification, if the formula is still not fully solved, the check function iterates through all possible assignments for the remaining variables using brute force. The algorithm checks if a satisfying assignment exists, prints whether the formula is satisfiable or unsatisfiable, and displays the assignment of variables. 3. DPLL algorithm a. If all clauses are satisfied with the current variable assignment, the solver returns the assignment as the solution. If any clause is empty, the solver returns unsatisfiable. Then, unit propagation is checked and implemented. If no unit clauses are found, the solver selects an unassigned literal and recursively tries to either assume that the literal is true, or backtrack and assume the literal is false. This checks all solutions until a satisfiable result is found or the formula is proven unsatisfiable.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. We used the following test cases: [[1, -2], [-1, 2]] [[1, 2], [-1, 2], [1, -2]] [[1, 2, 3], [-1, -2], [2, -3], [1, 2]]

	[[1, 2], [1, -2], [2, 3], [-1, 3], [-3, -2]] [[1], [-1]] We used these test cases because they range from simple 2-literal formulas to more complex 3-literal problems with a mix of positive and negative literals. We ran these problems by hand as well to check their correctness. The assignments can be different as there can be more than one satisfying assignment. The final case confirms that the solver can identify unsatisfiable problems. Together, these tests validate the correctness of the algorithm. All three codes give the same result, further showing they handle the formulas correctly.
12	How you managed the code development: We split the projects between all of us, so each person did one. We did little by little every day and merged everything together once we knew our scripts worked with basic test cases. We then created graphing scripts to make sure our codes worked properly. In each script, we took advantage of the use of functions and python libraries to simplify the process. We also followed the professor's DumbSAT code through our own development. The code development was rough, and some projects were harder than others, but we believe we were able to figure it out in the end as a team. Even though one person was in charge of each, we all helped each other when needed.
13	Detailed discussion of results: **Unit Clause SAT Solver** emerges as the fastest and most efficient overall. Its use of unit propagation allows it to quickly simplify problems, leading to much lower execution times, especially for satisfiable cases, and it remains effective even as the problem size grows. **DPLL**, while a more sophisticated algorithm with backtracking, performs well for satisfiable instances but starts to show significant performance degradation with unsatisfiable cases as problem size increases, making it a good middle-ground solution. However, **Incremental Search** is the slowest of the three, particularly for larger, unsatisfiable problems, due to its brute-force nature and lack of advanced pruning techniques. Incremental search is volatile, sometimes it performs very well and surpasses unit clauses on time, however other times it performs very badly.
14	How team was organized: The team was organized in a way that implemented both individual work and team work. Each person took charge of one of the projects and dove into deeply understanding its prompt. When we needed help, we helped each other develop our code, as well as explain any theoretical parts when needed. When graphing, we did everything together and made sure to meet many days to keep each other updated on the process.
15	What you might do differently if you did the project again: We would have taken more time to read all of the instructions and understand what the problem we were trying to solve is. Sometimes, Computer Science students rush too much on getting started to code, however it is utterly important to understand what the algorithm is set to do, how it should be done, and ultimately for example, guidelines such as test cases, inputs, outputs, etc.
16	Any additional material: N/A