Project 1 Readme Team AJC

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"

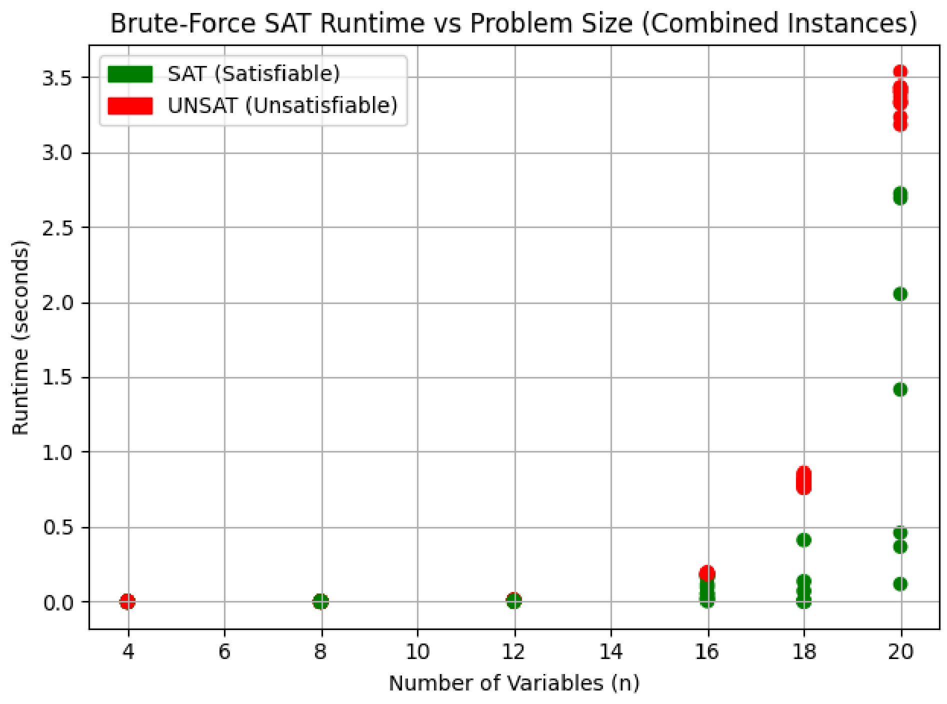
Also change the title of this template to "Project x Readme Team xxx"

1	Team Name: AJC
2	Team members names and netids: Andrew Cotaj (acotaj)
3	Overall project attempted, with sub-projects: SAT - Brute Force
4	Overall success of the project: 10/10
5	Approximately total time (in hours) to complete: ~11-12 hours
6	Link to github repository: https://github.com/AdrewC2026/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). Add more rows as necessary.

File/folder Name	File Contents and Use
Code Files	
root	├── plot_final_runtime_AJC.py ├── plot_runtime_AJC.py
Test Files	
input/	├── input │ ├── 2SAT.cnf │ ├── cnffile.cnf │ └── kSAT.cnf
Output Files	
results/	├── results │ ├── brute_force_2SAT_sat_solver_results_AJC.csv │ ├── brute_force_cnffile_sat_solver_results_AJC.csv │ └── brute_force_kSAT_sat_solver_results_AJC.csv
Plots (as needed)	
plots_AJC/	├── plots_AJC

	| | │ ├── bruteforce_runtime_2SAT_plot_AJC.png │ ├── bruteforce_runtime_cnffile_plot_AJC.png │ ├── bruteforce_runtime_kSAT_plot_AJC.png │ └── final_bruteforce_runtime_AJC.png |
8	Programming languages used, and associated libraries: python + matplotlib + re Excluding dependencies download in uv sync
9	Key data structures (for each sub-project): Throughout the implementation, several core data structures were used to support both the SAT solver and the analysis pipeline. The brute-force SAT solver relied heavily on Python dictionaries, most notably a Dict[int, bool] mapping each variable to a Boolean value to represent a single complete assignment during evaluation. The CNF formulas themselves were stored using lists of lists, where each clause was represented as a List[int] of literals and the entire formula as a List[List[int]]. When parsing DIMACS-like CNF files, each instance was packaged as a tuple containing its instance ID, number of variables, and clause list, and all instances were accumulated in a list for sequential processing. In the modified parser, tokenized lines were also handled as lists of strings after normalizing comma- and space-separated formats (hence importation of re library). For runtime plotting, the script collected data points into parallel lists for the x-values (number of variables), y-values (runtime), and point colors (based on satisfiability). Additionally, a defaultdict(int) was used to track how many instances were plotted for each n_vars, enabling an optional cap on the number of points per variable size to avoid visual clutter. Together, these structures provided an efficient and simple way to organize assignments, CNF representations, parsed instances, and experimental results throughout the project.
10	General operation of code (for each subproject): The general operation of the code follows a structured pipeline built on the provided project skeleton. When the program is executed, it reads the project configuration to determine that the SAT brute-force solver should be used, then loads one or more CNF instances through the DIMACS parser, which normalizes each file into a list of instances containing the instance ID, number of variables, and clause sets. For each instance, the SAT solver iterates through all possible truth assignments up to a defined cutoff (to prevent exponential blowup), constructing each assignment as a dictionary and checking it against the formula using helper functions that evaluate clauses and the full CNF. The solver records whether an instance is satisfiable, captures the runtime, and writes these results into CSV files through the helper infrastructure. Finally, the separate plotting script aggregates selected rows from multiple CSV files, filters them based on variable size, and generates a runtime visualization that highlights the exponential behavior of brute-force SAT. Overall, the code moves cleanly from configuration →

	parsing → solving → result logging → plotting, with each stage modularized within the project structure.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. To evaluate the correctness and behavior of my brute-force SAT solver, I used three main sets of test cases: the small cnffile.cnf provided with the project skeleton, as well as the larger 2SAT.cnf and kSAT.cnf datasets. The small file served as an initial sanity check - its low variable count allowed the solver to quickly evaluate all assignments, confirming that clause evaluation, CNF parsing, and CSV output were functioning correctly. The 2SAT.cnf and kSAT.cnf files contained many more instances across a wide range of variable sizes, which made them valuable for stress-testing both the parser and the solver. These datasets helped reveal exponential runtime behavior and validated that the solver produced consistent results across satisfiable and unsatisfiable formulas. They also confirmed that skipping large-n instances (beyond the brute-force cap) behaved as intended, preventing the solver from hanging on infeasible workloads. Together, these test cases demonstrated that the implementation was correct for small and moderate SAT instances, robust under larger inputs, and aligned with theoretical expectations for brute-force SAT.
12	How you managed the code development: I managed the code development by working incrementally and validating each component as it was introduced. I first ensured that the basic project skeleton ran correctly, then implemented and tested the brute-force SAT solver on small CNF files to confirm correctness before moving on to larger datasets. When expanding the DIMACS parser to handle different formats, I tested parsing independently before integrating it with the solver. I also added safeguards (such as a variable-count cap) to prevent exponential blowups during debugging. Throughout development, I repeatedly examined CSV outputs and plotted runtimes to verify that results were consistent and that the solver behaved as expected across different input sizes. This iterative, test-driven approach made it easier to catch issues early and maintain clean, working code at each stage.
13	Detailed discussion of results:



(Above in plots_AJC folder in repo)

The runtime plot clearly illustrates the exponential growth in computation time as the number of variables increases in brute-force SAT solving. For small instances (4–12 variables), both satisfiable (green) and unsatisfiable (red) formulas were solved almost instantly, with runtimes clustering near zero and showing no meaningful difference between SAT and UNSAT cases. This matches theoretical expectations, as 2^n remains small for these values, allowing the solver to enumerate all assignments quickly. As the number of variables increased to 16, the runtimes began to spread out, with unsatisfiable instances generally taking longer than satisfiable ones. This behavior makes sense because satisfiable formulas terminate early when a satisfying assignment is found, while unsatisfiable ones must exhaust the entire search space.

The exponential nature of brute-force search becomes dramatically visible at 18 and especially 20 variables. Around 18 variables, runtimes begin to exceed one full second, and by 20 variables several unsatisfiable instances reach well over 3.5 seconds. The steep vertical spread of points at n=20 reflects the fact that the solver must evaluate over one million possible assignments for each instance, and unsatisfiable cases consistently take the longest due to the need to test all possibilities. In contrast, some satisfiable formulas at n=20 still complete faster, showing the advantage of early termination. Overall, the plot demonstrates the classical exponential blowup of brute-force SAT and highlights why even modest increases in variable count rapidly become computationally infeasible—underscoring the limitations of brute-force search and the necessity of more advanced SAT-solving heuristics and pruning techniques for larger instances.

14 How team was organized:

	Individual Project - N/A
15	What you might do differently if you did the project again: If I were to redo the project, I would focus on improving flexibility and reducing manual adjustments throughout the code. In particular, many of the scripts rely on hard-coded file paths and filenames, which made it easy to lose track of where changes needed to be made as I switched between datasets. I would restructure the project so that input files, output directories, and configuration options are handled globally (either through a single config file or command-line arguments) so the solver and plotting scripts remain fully reusable without editing multiple files. I wasn't sure whether this would violate the project's structural requirements, so I avoided making large architectural changes, but in a real development setting I would definitely modernize this aspect for better maintainability.
16	Any additional material: What I learned: As I played around with the instances of different variable sizes, it became apparent just how fast and how dramatically the runtime scaled as the number of variables increased. I initially thought my program was stuck or had a bug, but I then came to realize that the solver was doing exactly what brute force requires: checking all 2^n possible assignments. Even modest increases from 16 to 20 variables produced massive jumps in runtime, which helped me internalize why brute-force SAT is considered impractical and why SAT is a canonical NP-complete problem.