Project 1 Readme Team EL

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: EL
2	Team members names and netids: Emmalyn Holmquist (ehomlqui) Lucia Raciti (lraciti2)
3	Overall project attempted, with sub-projects: SAT (best case, brute force)
4	Overall success of the project: success
5	Approximately total time (in hours) to complete: 7-9 hrs.
6	Link to github repository: https://github.com/eholmquist01/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	
src/sat_EL.py	Main SAT solver implementation of brute force and best case
graph_results_EL.py	Code for graph results showing the time it took for the instance vs. the number of variables used in an instance with each data point labeled as that instance and the coloring of the instances with solutions are marked in green whereas the instances with no solutions are marked in red.
Test Files	
module_tests/check_EL.py	Test script for checking the correctness of SAT solver functions using the cnffile file and the 2sat file
input/cnffile.cnf input/data_EL.cnf	CNF input file that contains multiple different types of instances with different number of clauses and variables and solvable and unsolvable solutions.
2SAT.cnf	CNF input file that contains problems with only two

	clauses, but multiple different amounts of variables and solution answers.
Output Files	
results/best_case_data_EL_sat_solver_results.csv	Output results from best-case SAT solver Contains the results of the SAT program ran with test files to display the instance id of the test, the number of variables used in the test, the number of clauses in the test case, what method of the sub-projects we used, whether the test has a solution, the time it took to run the program, and if there was a solution, the values that each variable is set to to get the solution. If there was not a solution, it contains the solution that would make the most clauses true.
results/brute_force_data_EL_sat_solver_results.csv	Output results of brute force SAT solver Contains the results of the SAT program ran with test files to display the instance id of the test, the number of variables used in the test, the number of clauses in the test case, what method of the sub-projects we used, whether the test has a solution, the time it took to run the program, and if there was a solution, the values that each variable is set to to get the solution.
Plots (as needed)	
plot_brute_force_results_EL.png	Graph of the brute force solver results of time for the instance vs. the number of variables for the instance. It also has coloring for problems with solutions (green) vs problems with no solutions (red)
plot_best_case_results_EL.png	Graph of the best case solver results of time for the instance vs. the number of variables for the instance. It also has coloring for problems with solutions (green) vs problems with no solutions (red)

8 Programming languages used, and associated libraries:
python:
- from typing import List, Tuple, Dict
- from src.helpers.sat_solver_helper import SatSolverAbstractClass
- import itertools
- import pandas as pd
- import matplotlib.pyplot as plt

9 Key data structures (for each sub-project):
Brute force & best case:
- List (of lists), tuples, dictionary,

	Graphing: - Pandas data frame
10	General operation of code (for each subproject) Brute force: used an iterator tool to generate all possible cases of true and false assignments for the variables from input csv that was already parsed in a for loop. In that loop, we created an assignment variable that was used to assign true or false to each variable. We then assigned the clauses as initially all true and went through each clause with the assigned variables. In a for loop, we then made each clause false until proven wrong through going through the assignment of variables in that clause and figuring out if the assignment would make the clause true or not. If the clause is made true, then we continue to the next clause. If it makes it false, then we don't continue with this possible combination and move on to the next assignment. If all clauses are true, then that assignment is a solution, and we return that there is a solution (True) and the assignment of variables in a dictionary that would make it true. If there is a clause that is not met, we go to the next assignment of variables, and if we reach the end of the for loop for iterating through all possible assignments, and there is still not an assignment that makes it true, we return false for having no solution, and we return an empty dictionary since there was no assignment that made it true. Best case: Implemented the same as brute force, but keep a tracker outside of the loop to count and keep track of the largest amount of clauses evaluated as correct, as well as a variable to keep track of the best case assignment. While in the loop, if we have an assignment that results in more true clauses then what the tracker holds, update the tracker and best case assignment variables, and continue on to the next possible assignment. If we encounter a clause that results in false, we will keep evaluating the rest of the cnf (as opposed to brute force), keeping track of the amount of clauses that result in true in case we need to update the tracker (if we have more true clauses then the tracker accounts for) and the best case assignment. Return these at the end.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. We had AI generate a test file that contained both satisfiable and unsatisfiable cnfs, ranging from 2-10 variables, with variable number of clauses, and 20 different problems. We chose this number because we wanted to test our code against a broad range of clauses and number of variables, as well as test going through multiple problems (having multiple problems would also generate a nicer graph for us). We chose to do under 10 variables so that the program wouldn't take a super long time to compile.
12	How you managed the code development: Used github to save changes to a remote repo. We weren't able to get the teams github function to work so we did all of the work on one computer and made commits/pushes throughout our process. We also began the project by brainstorming and exploring the repo before beginning with code development.
13	Detailed discussion of results: Our results are mostly shown through the graphs since the graphs show the time that an instance took vs. the amount of variables used in that instance, and each point is

	marked as solvable or unsolvable. According to the brute force graph, the instances with the least amount of variables tended to take the least amount of time. As more variables were added, for non-solvable instances, the shape of the graph was exponential as the instances with a lot more variables tended to take exponentially more time than those cases where there were only two or three variables. This is generally a similar pattern for both brute force and best case; however, in best case, the instances sometimes take longer (especially seen as the size gets bigger) because they continue to go through each clause (after encountering a clause evaluated to false) to keep track of how many clauses evaluate to true to find the best case. On the other hand, in brute force, the program exits out of the for loop immediately after finding one false clause and goes onto the next assignment. For the instances with solutions, the answers varied depending on how early the solution was found in the iterator. In our graph of the solvable instances, it looked to be linearly increasing as more variables were added, possibly because they went through more assignment options before finding a solution. However, this is not logically always the case since the first assignment for the instance could be true, we just did not have any test cases where that occurred.
14	How the team was organized: We worked together a few times in person to implement our best case and brute force code. We worked separately to brainstorm code for graphing and implementing test cases, and then got together to actually implement the code and work on the Readme file.
15	What you might do differently if you did the project again: We tried minimally to get the team's functionality to work on github. In the future, we might put more effort into getting this functionality to work so that we could collaborate remotely instead of having to find time to always work together on the code portion.
16	Any additional material: It took a minute to understand the structure of the repo; it wasn't very clearly explained past the point of downloading uv and forking/cloning. The hardest part of the project wasn't actually implementing brute force and best case, but figuring out about the helpers and how the program was functioning outside of our code. It also would have been helpful for the functions to have some docstrings explaining the input and output of the function.