Project 2-SAT Solver Readme Team Kelle-yeah!

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: Kelle-yeah!</td></tr>
<tr><td>2</td><td>Team members names and netids: Anna Kelley akelley5</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: 2-SAT Solver</td></tr>
<tr><td>4</td><td>Overall success of the project: successful</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 16</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/akelley04/TOC-fall-2024</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>2SAT_Kelle-yeah!.py</td><td>Main code DPLL algorithm 2SAT Solver</td></tr>
<tr><td colspan="2">Test Files</td></tr>
<tr><td>check-unitprop-Kelle-yeah!.csv</td><td>specific test to check unit_prop function</td></tr>
<tr><td>data_Kelle-yeah!.csv</td><td>final test taken from course website to determine accuracy</td></tr>
<tr><td colspan="2">Output Files</td></tr>
<tr><td>output_final_Kelle-yeah!.txt</td><td>printed output of code</td></tr>
<tr><td>output_xsat_Kelle-yeah!.txt</td><td>collected output for x-axis satisfiable expressions</td></tr>
<tr><td>output_xunsat_Kelle-yeah!.txt</td><td>collected output for</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td></td><td>x-axis unsatisfiable expressions</td></tr>
<tr><td>output_ysat_Kelle-yeah!.txt</td><td>collected output for y-axis satisfiable expressions</td></tr>
<tr><td>output_yunsat_Kelle-yeah!.txt</td><td>collected output for y-axis unsatisfiable expressions</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td>plots-2SAT_Kelle-yeah!.png</td><td>image of plot created in excel with estimated equation of bounding curve</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries:
a. language: Python
b. libraries: csv, time</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):
a. My two main data structures were for clauses, a list of lists containing ints, and values, a list of boolean values. I also made use of dictionaries, such as purevars in pure_elim function. Clauses: holds each clause and is passed through all the functions to eliminate the literals within. Values: this keeps track of each literal’s truth assignment. Because the literals are represented as numbers, the corresponding holder within this list is at index abs(literal) - 1. Other data structures, such as singles and purevars, are used to remember necessary literals while sweeping through the clauses or into each clause to look for a particular parameter, depending on the function.</td></tr>
<tr><td>10</td><td>General operation of code (for each subproject):
a. This program functioned by utilizing the DPLL algorithm. The three basic components of the DPLL algorithm are unit propagation, pure literal elimination, and the DPLL recursive call. For this particular project, there also had to be the added step of reading in a csv of a certain format. This is the first step of the program. Within this function, the timer is started, and we call our DPLL function and give it our collected clauses and a list of values set to None. Within DPLL, the function first looks to see if any clause in clauses has length 1–meaning the clause only contains a single unassigned literal. If yes, then we call unit_prop (explanation later). After this, function pure_elim is called. This deals with cases where a variable</td></tr>
</table>

	appears only as one polarity within the expression (explanation later). We then see our base cases. The first base case is when the list of clauses is empty. This means that the expression is satisfiable. The second base case is when a clause within clauses is empty. This means that the expression is unsatisfiable. If neither of these conditions are met, recursion is called. For recursion, two traversals are tried. To begin, a literal is chosen simply by using the first literal of the first clause. For the first attempted traversal, the clauses handed to DPLL is given the addition of our chosen literal. This works because it will force a unit propagation to occur and simplify from there. If our base case reaches an unsatisfiable condition, the program will backtrack and try a different traversal where we negate our chosen literal. Referencing back to unit_prop: Because the clause only has a single literal, the literal must be assigned whatever necessary value to make the clause true. Once a single literal clause has been found and the proper truth value assigned, the program removes any other clause, the entire clause, from clauses if literal appears in it and also if the negation of that literal appears, remove only the literal from each clause of clauses. Because of the while loop within DPLL, as long as unit clauses are found, the function will continue to be called. Reference back to pure_elim: The function begins by populating a dictionary. From there dictionary comprehension is used to filter out the pure literals. Once the pure literals are the only ones left in the dictionary, they are assigned the proper truth values. Then where any appearance of a pure literal exists, remove clause from clauses. Because this function takes out all pure literals, the function does not have to be looped.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. **a. check-unitprop-Kelle-yeah!.csv:** this data came from a doc called "CNF_file_format" provided on the course website. I modified this expression example to be used for a 2-SAT solver. Specifically, when starting my code, I tested by condition starting with unit propagation. This check file is to test what the outcome truth values is for just unit propagation. Based on the truth values, I could see how different negations affects others. For a final check, I used an online website to verify the outcome of a few of the expressions to ensure they agreed with mine.
12	How you managed the code development: a. I began by writing code to work to read in a csv based on the code snippet in "CNF_file_format". I then followed the Wikipedia page closely to be the general structure for my DPLL function. I knew then I would need a function to test conditions for unit propagation and a function for

	pure literal elimination. To better understand the effects of each function on an expression, I wrote examples by hand and crossed out literals and clauses when necessary. Once my functions were written, I tested by modifying the given csv with different negations and variables. I also talked through my code with a TA.
13	Detailed discussion of results: a. From the course-given test of 100 different expressions, I received 50 satisfiable and 50 unsatisfiable, which was expected based on class discussions. The program is accurate to determine satisfiability for 2SAT problems. Many of the results were checked by a second source website.
14	How team was organized: a. The team began on the code first and once complete filled out all the necessary documents.
15	What you might do differently if you did the project again: a. I would like to improve the code for the program to run faster. I also could improve the output and possibly for a smaller dataset show the process of each decision.
16	Any additional material: N/A