Project 1 Readme Team hberens

Version 1 9/11/24

<table>
<tr><td>1</td><td>Team Name: hberens</td></tr>
<tr><td>2</td><td>Team members names and netids
Helena Berens, hberens</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: I attempted to implement the DPLL algorithm in a k-SAT solver</td></tr>
<tr><td>4</td><td>Overall success of the project: It works as expected and solves for satisfiability quickly using the DPLL algorithm and graphs everything correctly.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: ~15 hours</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/hberens/theory_project1_hberens</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>DPLL_SAT_hberens.py</td><td>The DPLL_SAT_hberens file reads a .cnf file and parses each WFF. It then runs a function to do the DPLL recursive backtracking algorithm used to determine satisfiability of CNF formulas. It times this algorithm for each WFF and then returns a csv file with that filename, execution time, number of variables, and satisfiability. It is used to test the DPLL algorithm and produce output.</td></tr>
<tr><td>plotting_SAT_hberens.py</td><td>This function reads the CSV file, processes the data, and generates a scatter plot that shows the execution time for each experiment</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td></td><td>based on the number of variables and whether the instance was satisfiable or unsatisfiable. It is used everytime we want to run the code to generate a plot.</td></tr>
<tr><td colspan="2">Test Files</td></tr>
<tr><td>data_kSAT_hberens.cnf</td><td>The CNF form file for kSAT WFF's. It has all the data we want to read in and test and plot. It is read in by the DPLL_SAT_hberens file.</td></tr>
<tr><td colspan="2">Output Files</td></tr>
<tr><td>output_hberens.csv</td><td>This is the CSV file produced with the filename, execution time, number of variables, and satisfiability for all the WFF's in the cnf file. It is produced by the DPLL_SAT_hberens file and then read in and graphed by the plotting_SAT_hberens file.</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td>plots_hberens.png</td><td>This is the scatter plot file that is generated when we run the plotting_SAT_hberens file and is saved in the directory so that we can look at it repeatedly.</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries:
Python
- csv
- matplotlib
- numpy
- time</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project): arrays
1) Clauses Lists (all_clauses and current_clauses)- These store the clauses for each well-formed formula (WFF) from the CNF file. Each clause is a list of integers where positive values represent variables and negative values</td></tr>
</table>

	represent their negations. The current_clauses temporarily holds the clauses of a single WFF while parsing. The all_clauses stores all parsed WFFs, each represented as a tuple of (num_vars, num_clauses, current_clauses) for easier access. 2) Assignment List (assignment)- This tracks the assignments of truth values to variables in the DPLL algorithm. During recursive calls, this list gets appended with either positive or negative values of variables to represent true or false assignments. 3) Results List (results)- This stores the results of each WFF evaluation for later export to a CSV file. Each result is a tuple containing: the filename, execution time, number of variables, whether the formula is satisfiable (S) or unsatisfiable (U). This list is populated after running the DPLL algorithm on each WFF and is then saved to a CSV file for reporting. 4) Plotting lists (num_vars_list, elapsed time list, satisfiable_list)- these 3 lists in the plotting file are used to store the number of variables in each experiment from the CSV file, to store the execution time for each experiment in seconds, and to track the satisfiability status of each experiment (either 'green' for satisfiable or 'red' for unsatisfiable).
10	General operation of code (for each subproject) 1) File parsing (parse_cnf_file): The program reads a .cnf file and parses each WFF. It looks for lines starting with p cnf to extract the number of variables and clauses. It stores the clauses (as lists of integers) until it encounters a comment line (starting with c), which signifies the end of a WFF. All WFFs are stored in all_clauses. 2) DPLL Algorithm (dpll): This is a recursive backtracking algorithm used to determine satisfiability of CNF formulas. It picks a variable, assigns it a value (either true or false), simplifies the formula, and then recursively checks if this assignment leads to a solution. If no solution is found, it backtracks and tries the opposite assignment. The algorithm returns True if the formula is satisfiable, along with the variable assignments. The input to this function is a list of clauses and the current assignment. It uses a base case to check if the clauses is empty and if so the formula is satisfied and we return true. If any clause is empty, the formula is unsatisfiable and we return False. After the base case, we select a variable, try assigning it True, then simplify it by removing clauses where this variable appears positively and eliminating the negated form from other clauses. We recursively call the dpll function on the simplified formula. If assigning `True` leads to SAT, return that result, and if not, we backtrack and attempt to assign False with the opposite simplification. This function returns a tuple of whether the formula is satisfiable and the assignment. It is efficient for many SAT problems because it narrows down the search space by eliminating clauses and literals at each step.

	3) Timing and Evaluation (run_dpll_and_time): For each WFF, the dpll function is called with an empty initial assignment. The time taken for this execution is recorded and computed. After evaluation, whether the formula was satisfiable or unsatisfiable is also captured. These results are returned. 4) Saving results (save_to_csv): After processing all WFFs, the results (execution time, satisfiability, number of variables) are written to a CSV file for further analysis or reporting. We are able to see the columns of the CSV file as filename, execution time, number of variables, and satisfiability. 5) Plotting results (plot_results): This function reads the CSV file, processes the data, and generates a scatter plot that shows the execution time for each experiment based on the number of variables and whether the instance was satisfiable or unsatisfiable. It uses csv.DictReader and appends all the column data from the CSV file into the various lists. It then uses the matplotlib to create a scatter plot figure with label, axis ticks, a title, a legend, a grid, and colors each point as a green circle for Satisfiable and a red x for Unsatisfiable. It then saves this plot to the plots_hberens.png file. This program provides a visual representation of the time complexity of solving SAT problems with the DPLL algorithm by plotting execution time against the number of variables and marking whether the instance was satisfiable or not ($O(2^n)$)
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. - I didn't add any specific test cases, however I was able to verify my output by working with smaller subsections of the kSAT cnf file. Since we know whether each wff would be S or U, I was able to check this on my graph as I went through one by one. I also managed to get a graph that was exponential and bounded by red x's which was what was expected. The cnf file also had a variety of sizes for the WFF's so I was able to check with many different cases. - I went through and hand wrote some test cases and traced through my code to verify it was working as I wanted it to. I was able to make sure I was getting the correct assignments this way, and then the graph proved by execution time was working correctly as well.
12	How you managed the code development - To manage code development, I broke up the tasks into smaller portions. First I studied the cnf file format and implemented a function to parse through the WFFs in this file. Then I researched the DPLL algorithm and repeatedly attempted to implement it. This was the main and most challenging portion of the coding. During this, I tested it with smaller WFF's and hand written test cases. Then after implementing the algorithm, I created a smaller function to run the dpll on the clauses I parsed from the cnf file and timed it's execution. Lastly, I implemented a function to save all this data to a CSV file as my output file to eventually plot. The last portion of my project was to create a file for plotting, and to do this I researched the matplotlib Python library and all the features that I

	could use in my graph. I parsed through the data in my output CSV file and designed a scatterplot as I saw fit. - To test my code, as discussed earlier, I used smaller hand-written test cases that I knew should be either satisfiable or unsatisfiable and checked my algorithms as I went. This helped me test both the correctness of the DPLL implementation and the structure of the output CSV. I also tested several edge cases to verify the handling of empty clauses, which should be unsatisfiable, trivial clauses (single literals), where the result could be determined without recursion, and larger formulas, to ensure the recursive depth and backtracking were handled correctly. I then used the files provided by Professor Kogge to generate large test cases for more extensive testing to ensure consistency. In addition to these test cases, I debugged my code as usual by adding print statements throughout to manually track the recursive and output and CSV file. Lastly, I compared my output on the graph to known SAT solvers as I knew what kind of graph I was expecting. Since I was getting an exponential graph bounded mostly by unsatisfiable points, I was able to confirm that everything was working correctly.
13	Detailed discussion of results: - The results of my timing produces an exponential graph in the plots_hberens.png file for the unsatisfiable wff's with all the satisfiable points bounded above by this line. The x-axis maps the number of variables in the wff, and the y-axis has the execution time of the solver. I go up to 25 variables and change by about 2 variables each time, and each one has numerous points for each size WFF. A red x indicates that the wff was unsatisfiable, and a green • indicates that it was satisfiable. This is the only graph I have since it plots all the wff's tested at once and I did this project individually, so I only did one project of the options. - The time complexity of the program is at worst O(2^n). Since the DPLL algorithm has a recursive nature, and works by selecting a variable, assigning it a truth value, and then simplifying the problem, in the worst case, it branches on every possible assignment of variables. This leads to a decision tree with 2 branches (T and F) for every step of the process. With n variables, the worst case scenario would then have 2^n branches. This is why we have an exponential growth for the unsatisfiable WFF's as shown on our graph. This is characteristic of NP-complete problems like SAT, where finding a solution can require checking all possible assignments. - While at worst it is O(2^n), the use of unit propagation and pure literal elimination can reduce the execution time because they help reduce the number of clauses and simplify the problem in recursion as well as remove pure literals early. This reduces the number of branches needing to be explored, so that is why our coded algorithm is still relatively fast. Additionally, the order in which variables are chosen can dramatically affect the efficiency, which for us is random, however that is why we see some faster or slower outliers at times on the graph.

	Thus in smaller problems or in satisfiable problems, we typically see the time complexity being less than 2^n
14	How team was organized- I worked on this by myself, so there was no organization that went into allocating work to teammates.
15	What you might do differently if you did the project again If I were to redo the project, I wouldn't attempt the incremental approach that I initially took and instead just start with the DPLL algorithm I submitted. Also, I ended up having to restart because I was generating random WFFs and producing a CNF file. So instead, I would skip that step and directly focus on parsing a CNF file from the start. I'd also aim to optimize the DPLL algorithm by incorporating more efficient variable selection strategies and conflict-driven clause learning (CDCL) for better performance. Additionally, I'd work on testing with larger, more complex SAT problems and restructure the code to make it easier to maintain. These changes would simplify the process and make the solver more efficient.
16	Any additional material: I attempted many different implementations of the k-SAT solver, but this was my most successful and quickest attempt, and thus the project I decided to submit. I did not generate any extra test scripts that still exist. So there is no extra material to be looked at. ** Important Note: My DPLL_SAT_hberens.py file already contains reading in the cnf file and outputs a CSV file. My plotting_SAT_hberens.py then reads in that CSV file called output_hberens.csv and plots it and saves this figure into a file called plots_hberens.png. Thus, to run the program, the TA can have it run the test data script by just calling ./DPLL_SAT_hberens.py and then immediately running ./plotting_SAT_hberens to generate the plot. They can then see the output in the CSV file and the plot in the png file.