Readme_teamname

Version 1 8/22/24

A copy of this file should be included in the github repository with your project. Change the teamname above to your own

1. Team name: Kelle-yeah!
2. Names of all team members: Anna Kelley
3. Link to github repository: https://github.com/akelley04/TOC-fall-2024
4. Which project options were attempted: 2-SAT Solver
5. Approximately total time spent on project: 16 hours
6. The language you used, and a list of libraries you invoked.
    a. language: Python
    b. libraries: csv, time
7. How would a TA run your program (did you provide a script to run a test case?)
    a. To run this program, ensure data_Kelle-yeah!.csv is in the same folder as 2SAT_Kelle-yeah!.py. Type into the command line "./2SAT_Kelle-yeah!.py". "data_Kelle-yeah!.csv" is hard coded into the main function of the code.
8. A brief description of the key data structures you used, and how the program functioned.
    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.
    b. 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 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.

9. A discussion as to what test cases you added and why you decided to add them (what did they tell you about the correctness of your code). Where did the data come from? (course website, handcrafted, a data generator, other)
    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.
10. An analysis of the results, such as if timings were called for, which plots showed what? What was the approximate complexity of your program?
    a. **plots-2SAT_Kelle-yeah!.png** shows the satisfiable expressions (green) and the unsatisfiable expressions (red) plotted with the x-axis as number of literals (calculated as number of clauses*2) and the y-axis as time. The bounding curve is for the worst case execution time and the approximation for the equation can be seen on the image.
    b. The approximate time complexity of the program is O(V). Although DPLL is (O2^V) because of its recursion, 2-SAT for most cases is linear.
11. A description of how you managed the code development and testing.
    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.
12. Did you do any extra programs, or attempted any extra test cases
    a. No