README_cander35

Version 1 8/22/24

This repository holds the contents for my Theory of Computing Project 1 that Rewrites DumbSAT to use an incremental search through possible solutions.

1. Team name: cander35

2. Names of all team members: Cate Anderson

3. Link to github repository: https://github.com/cateranderson/TOC_Project1.git

4. Which project options were attempted: Rewrite DumbSAT to use an incremental search through possible solutions

5. Approximately total time spent on project: Quite long - maybe 20 hours

6. The language you used, and a list of libraries you invoked.: Python- time, csv, sys, matplotlib.pyplot, numpy

7. How would a TA run your program (did you provide a script to run a test case?): make sure the script and input file of choice is in the same directory.
   python3 name_of_code.py input_file_name.csv desired_output_file.txt
   ex:
   python3 DumbSATIncrementer_parseCSV_plotData_cander35.py kSAT_input_cander35.cnf.csv output_cander35.txt

8. A brief description of the key data structures you used, and how the program functioned.:
   Key Data Structures:

- List:
  - Problems list: Stores the parsed CNF (Conjunctive Normal Form) problems from the input. Each problem consists of a tuple containing the WFF (well-formed formula), the number of variables (Nvars), and the number of clauses (Nclauses).
  - WFF list: Stores individual clauses, where each clause is itself a list of literals (integers representing variables or their negations).
  - Clauses list: Inside WFF, each clause is represented as a list of integers (positive integers for variables and negative integers for negated variables).
- Dictionary:

- Assignment dictionary: Holds the current truth assignment for each variable. The key is the variable number, and the value is 1 (True) or 0 (False).

How this program functions:

- Parsing CNF Problems (parse_cnf_csv):
  - The program reads a CSV input file (or stdin) containing multiple CNF problems.
  - It skips comments and identifies the CNF format via the p cnf line, extracting the number of variables (Nvars) and clauses (Nclauses).
  - Clauses (sets of literals) are stored in lists, and each problem is appended to the problems list as a tuple: (wff, Nvars, Nclauses).
- Checking Clause Satisfaction (is_clause_satisfied):
  - For each clause, the program checks if at least one literal satisfies the clause.
  - For positive literals, if the corresponding variable is assigned True (1), the clause is satisfied.
  - For negative literals, if the corresponding variable is assigned False (0), the clause is satisfied.
- Incremental SAT Solver (incremental_sat):
  - A recursive depth-first search tries to assign truth values to variables.
  - For each variable, it first assigns True (1) and recursively checks if this leads to a satisfying assignment for all clauses.
  - If not, it backtracks and tries assigning False (0).
  - The base case is reached when all variables are assigned, and it checks if all clauses are satisfied.
- Testing a WFF (test_wff):
  - For each problem (WFF), the solver tests whether a satisfying assignment exists.
  - It records the execution time taken to either find a satisfying assignment or determine the problem is unsatisfiable.
- Logging Results (log_results):
  - The results for each problem (problem number, number of variables, satisfiability, execution time, and the satisfying assignment if applicable) are written to an output file or standard output.
- Plotting the Results (plot_data):
  - After solving the problems, the program reads the output file and plots the number of variables against the execution time.
  - Satisfiable problems are marked with green circles, and unsatisfiable ones with red triangles.
  - A line of best fit representing the worst-case complexity (2^n) is also plotted.

9. <u>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)</u>:

I got my data from the course website. I decided to add the kSAT.cnf.csv test case (which I renamed to kSAT_input_cander35.cnf.csv) because it provided a wide range of problem variations—400 in total—with diverse variable counts. This variety helped me evaluate the scalability and performance of my code across different problem sizes and complexities.

By including this test case, I could verify how well my algorithm handled cases with varying clause lengths and numbers of variables, which are critical for testing the robustness and general applicability of my solution to kSAT problems. The more verbose nature of this dataset allowed me to confirm that the code could efficiently process large inputs without significant slowdowns or memory issues.

Moreover, the test case helped me identify specific edge cases, such as those with minimal clauses or an overwhelming number of variables, ensuring that the implementation remained correct under both trivial and complex conditions. The correctness of the code was validated by comparing the output to expected results provided by the course, and the large dataset made it possible to observe any trends or inconsistencies in the code's behavior over numerous test runs.

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?:

The approximate complexity of my program was O(2^n). This is because the overall complexity is O(P * 2^Nvars * Nclauses), where P is the number of problems, Nvars is the number of variables, and Nclauses is the number of clauses per problem. The exponential complexity arises because the algorithm checks every possible truth assignment (which grows exponentially with the number of variables) in order to find a satisfying assignment.

My output generally demonstrated that the greater the number of variables, the longer the computation times were. Additionally, the greater the number of variables, the time it took to reach a result of unsatisfactory took much longer than with a smaller number of variables. This highlights the exponential growth in difficulty as the number of variables increases.

I feel that the plotted data demonstrates this pattern quite well. The plot illustrates the performance of the SAT solver by depicting the relationship between the number of variables and the execution time in microseconds. On the x-axis, the number of variables is plotted, while the y-axis shows the corresponding execution time. Two types of problems are represented: green circles indicate satisfiable problems, and red triangles denote unsatisfiable ones. Additionally, a dashed blue line highlights the theoretical worst-case time complexity, which follows an exponential O(2^n) growth pattern, illustrating the expected performance for unsatisfiable problems. This plot provides a clear visual comparison between the actual performance of the solver and the theoretical limits, demonstrating how the execution time scales as the problem size increases. You can see that as the number of variables increases, the amount of red triangles depicted also increases at a higher y-axis location.

11. A description of how you managed the code development and testing.:

To tackle this incremental search approach, I focused on several key steps to enhance efficiency. Instead of generating all possible assignments upfront, I decided to build partial assignments clause-by-clause. If a partial assignment failed to satisfy a clause, I backtracked immediately, avoiding irrelevant combinations.

I utilized a recursive function that explores potential solutions in a depth-first manner, setting one variable at a time and checking its consistency with the constraints. If a partial assignment was valid, I proceeded; if not, I backtracked.

Key changes in my approach included replacing the complete enumeration of assignments with this recursive function, which tests each variable in both possible states (True or False) incrementally. I also implemented an early exit strategy, stopping the search once a solution is found, which reduced the number of assignments checked. Lastly, I ensured that if any partial assignment failed to satisfy a clause, I backtracked right away, further streamlining the process.

Additionally, I started testing with smaller self made examples that I knew the solutions to. Once I knew this worked, I started the csv process and had code that parsed simple csv files I made. Then once this worked, I started importing Dr. Kogge's large test files. Once I got this to work and had my output formatted the way I thought was most useful and readable I started working on parsing the output to create a useful plot that could inform users on the output of large csv files with many different variables and satisfaction outcomes.

12. Did you do any extra programs, or attempted any extra test cases:

Not necessarily. Besides making some simple input files to test my code, I didn't use many other test cases besides the one I have in my repository and some other files Dr. Kogge provided. I have many scripts that I would add to in order to grow and develop my code to have more functionality, but the goal was to develop the final script that I submitted, I didn't create separate programs that were unrelated to the end goal.