# Project Readme Template

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: Trigonometry</td></tr>
<tr><td>2</td><td>Team members names and netids: Ella Trigiani etrigian</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: SAT Brute Force</td></tr>
<tr><td>4</td><td>Overall success of the project: I think it was successful, my output looks correct and it runs in good time.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 10 hours</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/ellatrigiani/Project1-TOC.git</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).
<table>
<tr><th>File/folder Name</th><th>File Contents and Use</th></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td>sat.py/src</td><td>
<ul>
<li>Class Definition (SatSolver) was given to us and provides infrastructure for file I/o, CSV writing, and plotting</li>
<li>Brute force solver (sat_bruteforce)
<ul>
<li>Takes n_vars, which is the number of boolean variables, and clauses as input</li>
<li>It returns tuple saying if the CNF is satisfiable and what the solution is</li>
<li>It performs edge case checks, just in case the test input calls for them
<ul>
<li>These return true or false if any clause is empty</li>
</ul></li>
<li>Creates an assignment dictionary mapping variables numbers (0s and 1s) to true and false</li>
<li>Calls recursive helper function (incremental_sat)</li>
</ul></li>
</ul>
</td></tr>
</table>
</td></tr>
</table>

| | |
|---|---|
| | ○ Converts internal format (0s and 1s) to output format (True and False) using dictionary comprehension<br>● Checking Clause Satisfaction (is_clause_satisfied)<br>○ For each literal in a clause, check if it is satisfies the current assignment<br>○ Positive literals are satisfied when the variable is true<br>○ Negative literals are satisfied when the variable is false<br>○ returns true if at least one literal in the clause is satisfied<br>● Checking formula satisfaction (formula_satisfied)<br>○ Iterates through all clauses in the formula<br>○ Returns false immediately if any clause is unsatisfied<br>○ Returns true only if all clauses are satisfied<br>● Incremental SAT solver (incremental_sat)<br>○ Recursive DFS through all possible assignments<br>○ Base case: when depth > n_vars, all variables are assigned, check if the formula is satisfied<br>○ Recursive case: trying to assign current variable to true then recursively solve for remaining variables<br>○ If true doesnt work go back and try false<br>○ If neither works remove the assignment and return false<br>○ Key characteristic of brute force: only checks formula satisfaction when all variables are assigned, rather than on partial assignments<br>○ Explores all $O(2^n)$ assignments |
| sat_runner_trigonometry.py/scripts | ● Path setup<br>○ Needed this root to the |

| | | |
|---|---|---|
| | | python path to enable imports from src/, otherwise it was failing<br>○ Allows script to be run from anywhere while accessing helper functions<br>● Importing helper functions<br>○ Needed to parse CNF files<br>○ Import SatSolver class to call the brute force solving method<br>○ Imports matplotlib and numpy so I could make the graphs<br>● Writing results to a CSV (write_results_csv)<br>○ Creates output directory if it DNE<br>○ Opens CSV file and writes the proper header<br>○ For each parsed instance we see the time of the execution, if it is satisfiable, and the solution<br>○ Collects timing data and variable counts for plotting<br>○ Returns lists needed to make graph<br>● Generating Plot (plot_from_results)<br>○ Creates scatter plot with number of variables (x_axis) vs execution time in seconds (y_axis)<br>○ Separates data points by satisfiability<br>○ Plots worst case time complexity<br>○ Demonstrates exponential growth increase<br>○ Saves as a png file<br>● Main<br>○ Parses command line arguments<br>○ Uses provided parser to extract instances from input file<br>○ Creates SatSolver instance<br>○ Calls CSV writing function to solve all instances and record results |

| | |
|---|---|
| | ○ Generates performance plot in same dictionary as CSV output<br>○ Provides command-line interface: uv run scripts/sat_runner_trigonometry.py input/data_trigonometry.cnf results/output_trigonometry.csv |
| Test Files | |
| cnffile.csv/input | ● Test file provided to us in CNF format |
| data_trigonometry.csv/input | ● Smaller version of the 2SAT file given to us, still over 1000 lines, but easier to run in an efficient amount of time to see the output |
| Output Files | |
| brute_force_cnffile_sat_solver_results.csv/results | ● Output file produced from the test input provided to us<br>● Automatically created when we run uv run main.py<br>● Stored in the results directory<br>● Outputs in a CSV file format with the correct columns we were given |
| output_trigonometry/results | ● Same gist as the previous output file, but this is run on a separate input I put in<br>○ I got the input from the 2SAT file on canvas but only took the first ~1000 lines because I wanted it to be large, but not take forever to run<br>● Outputs a CSV that highlights satisfiability in the same order as the previous output<br>● Ran on a lot more CNFs so much larger than the previous file |
| Plots (as needed) | |
| output_graph_trigonometry.png/results | ● Generated from plot_from_results in the sat_runner_trigonometry.py<br>● Generated after solving for all instances and writing the CSV |

| | | |
|---|---|---|
| | | outputs<br>● Saved as a PNG image in the same directory as CSV output<br>● Graph components<br>○ X-axis is the number of variables which shows how complexity increases with more variables<br>○ Y-axis is the time in seconds, which shows the actual runtime of the brute force algorithm<br>○ The green circles are the satisfiable problems<br>○ The triangles are the unsatisfiable problems<br>○ The dotted blue line is the theoretical complexity of $2^n$, which we can see the points sort of follow the line<br>● This graph very clearly displays that as the number of variables increase the runtime grows exponentially |
| 8 | Programming languages used, and associated libraries: I used Python as the only programming languages and then for libraries I used:<br>1. Typing - which helped with cleaner code documentation. This was also provided to us on the skeleton file.<br>2. Time - which I used in my own csv parsing function, since I had some difficulty patching the helper function into my local runner.<br>3. Sys - used to command line argument parse and path manipulations<br>4. Os - also used for directing paths and files<br>5. Matplotlib.pyplot - creating scatter plots and visualizations<br>6. Numpy - to do mathematical operations to plot the complexity curve | |
| 9 | Key data structures (for each sub-project):<br>1. List of lists (clauses) - The CNF formula is represented as a list of clauses, where each inner list is a clause containing literals. Positive integers represented variables and negative integers represented negated variables.<br>2. Dictionaries (variable assignment) - Primary use in the main algorithm was mapping variable numbers to truth values.<br>3. Tuple (return values) - All the solver methods return a tuple, where the first element is the satisfiability status and the second is the solution dictionary.<br>4. Lists - used to parse instances stored in the format of the output csv header which allowed processing multiple CNF problems from a single file. | |

| | |
|---|---|
| 10 | General operation of code (for each subproject)<br>Description of sat.py (main code)<br>1. I needed to parse the input, but luckily this was given to us in the repo, under helper functions.<br>2. Next I wanted to make sure I handled any possible edge cases considering I was unsure what the test input would be.<br>3. Next I did a recursive search in the incremental_sat() function starting at variable 1 with an empty assignment.<br>4. Then I had to iterate through the number of n_vars and perform the variable assignment. I would first assign true, then recursively solve the remaining variables. If it failed I would backtrack and try False. If both failed, I would remove the assignment and return false.<br>5. Since I used recursion I needed a base case which checked when all variables were assigned, if the formula was satisfied.<br>6. Then I needed to make sure to check the clauses and see for each clause if there was at least one literal that was true under the assignment<br>7. Then I needed to check the formula and make sure all the clauses were satisfied.<br>8. Then I converted all the 0s and 1s into True and False in dictionary format for the output.<br>9. Then lastly I returned a tuple of (satisfiability, solution_dict)<br><br>Description of sat_runner_trigonometry.py (local runner I used to output graphs and test other inputs):<br>1. I first added the project root to Python paths for imports, since without this piece I was getting a lot of errors. To be totally honest, I found the method for doing this online by reading what the error meant and how to solve it.<br>2. Next I parsed the input calling on the parser given to us in the helper functions.<br>3. Then I called the solver that I wrote, with the timer, and then captured the results as rows for the CSV.<br>4. Then I made a function to save all the results as a CSV, which I don't think I needed to do but honestly I was having trouble piecing it with the helper one for some reason, and it was pretty simple to write.<br>5. I then made sure to capture all needed info from the CSV for the plot.<br>6. I then generated the plot to show the exponential relationship between the number of variables and the time.<br>7. I then saved the outputs of the csv and png. |
| 11 | What test cases you used/added, why you used them, what did they tell you about the correctness of your code.<br>I used two test cases, the first being the input we were given called cnffile.cnf and then the other was a portion of one of the files posted on canvas which I called data_trigonometry.cnf. The first one was simpler to work out, since there were only 3 instances. But by running through the logic on paper, it matched what my code outputted, leading me to believe it was a |

| | |
|---|---|
| | success. For the other one it was much longer, so I used more of a spot check to assess correctness. But since I knew my code worked on the first ones, I was confident it would also work on the longer test script. One of the most helpful things actually was the graph, since it showed the relationship made sense between the number of variables and the runtime. |
| 12 | How you managed the code development: I approached this project in phases. The first phase was to understand the problem. I had to become comfortable working within the classes and with the CNF files. I initially struggled a lot with the import paths and how to piece everything together. The second phase was writing the algorithm, which took awhile considering there were a bunch of pieces to cover. I added many helper functions as well as the main recursive function. The third phase was figuring out how to run the main.py and get that all to work. I had lots of trouble with import path issues, since everything was in different folders. The fourth phase was creating the script to read CSVs and plot, which again I know was silly to make another csv read function but honestly trying to patch it with the helper one was causing so many issues. The last phase was verifying solutions and making sure everything worked and looked okay. |
| 13 | Detailed discussion of results:<br><br>The brute force SAT solver has a time complexity of $O(2^n)$, where n is the number of variables. For example a test with 4 instances, executing from times between 11 to 41 microseconds since only 16 assignments need checking. But the more variables there were the more this slowed down. There was also a clear performance difference between problems that were satisfiable versus unsatisfiable. Satisfiable problems solved much quicker since they terminated once a solution was found, but unsatisfiable instances took much longer since all $2^n$ possibilities had to be explored. This pattern is evident on the performance plot.<br><br>All the solutions were either manually verified for correctness or spot verified for correctness. So I am certain the code runs effectively, as mentioned above the performance plot validates that there is an exponential relationship between problem size and run time. Brute force, however, is still efficient because the test cases used weren't too big. I think at scale, if a company implemented this, it would be smart for them to explore other options. One positive of brute force is its thoroughness, it is easy to know if the program is working correctly. Overall the results were very positive and insightful. |
| 14 | How the team was organized: I was working independently on this project, so this is not really applicable. |
| 15 | What you might do differently if you did the project again:<br><br>If I were to approach this differently, if I could do it over again, I would:<br>1. Start earlier, which would give me more time to understand the infrastructure before implementing the algorithm.<br>2. Read all documentation thoroughly first, to understand the file structure and the use of main.py.Some of the instructions and I didn't exactly know what to do, so taking a more thorough approach would have helped clarify the confusion.<br>3. Separate concerns better, like creating a standalone test script early on to verify my |

| | |
|---|---|
| | algorithm was working. I did this more towards the end, but it probably would've been smarter to do it in the beginning.<br>4. Lastly, definitely ask for clarification sooner. Some aspects of the file format, what we were supposed to produce, and expected output weren't immediately clear, but other than that it was good. |
| 16 | Any additional material:<br><br>Not really. I was just going to add that I think it would be better for the instructions to be a bit clearer. There are a lot of moving parts and it can get a little messy to understand. But overall, I do think this helped me learn a lot since it was complicated: both about SAT solving and general Python usage.<br><br>Also just wanted to point out again, in my own personal runner the reason I wrote a csv parsing file was because calling the helper function given to us was causing a lot of problems on my end. I also didn't want the file to be named what the parser the TA assigned names the output file, but couldn't edit that because it is needed for the autograder. It also was pretty simple to implement. |