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: Surgical</td></tr>
<tr><td>2</td><td>Team members names and netids: Sean Quigley – squigle2, Trey Cotey – dcotey, mlajoie</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: SAT (brute force, backtracking, best case)</td></tr>
<tr><td>4</td><td>Overall success of the project: Great! We all felt confident in our contributions and results, as well as learned a good deal about this fundamental computer science problem.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 8 hrs</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/SeanQuigley1/Project1-TOC</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>src/sat_surgical.py</td><td>Our code for brute force, backtracking, and best case SAT. We also have a verifier helper function used to determine if assignments work, don't, or need to be explored further (i.e. assign more literals). The results are stored in results</td></tr>
<tr><td>src/sat_graph_surgical.py</td><td>This is a script that uses matplotlib to separately plot the time statistics of each algorithm approach (number of literals x time). These graphs are stores in results</td></tr>
<tr><td>src/entry_point.py</td><td>Modified entry_point to orchestrate communication between sat_solver_helper and sat_graph_surgical (i.e. send results from sat_solver_helper to sat_graph_surgical)</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td>src/helpers/constants.py</td><td>Added various input file names to be swapped out for testing purposes</td></tr>
<tr><td>Src/helpers/sat_solver_helper.py</td><td>Adjusted the code to return the results from the various sat algorithms it runs to entry_point</td></tr>
<tr><td colspan="2">Test Files</td></tr>
<tr><td>Input/cnffile.cnf</td><td>Original input/test file given</td></tr>
<tr><td>Input/data_2SAT_surgical.cnf</td><td>Additional input file to test larger clause sizes (from resources in canvas)</td></tr>
<tr><td>Input/data_kSAT_surgical.cnf</td><td>Additional input file to test larger literal sizes (from resources in canvas)</td></tr>
<tr><td colspan="2">Output Files</td></tr>
<tr><td>Results/~.csv</td><td>Files containing the results of each SAT algorithm for a specific input file (each subproblem has its own csv per input file)</td></tr>
<tr><td>Results/output_verification_uw_sat_solver_surgical.pdf</td><td>Screenshots demonstrating verification process used with University of Washington Online Sat Solver against our algorithms</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td>Results/~.jpg</td><td>Plots for number of literals x time of each SAT sub problem (one plot per subproblem per input file)</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries: We developed using Python and the libraries made available to us for the SAT subproblems. For the graphing portion, we used matplotlib and os (for file names/file paths / saving files). Matplotlib was used to create standard scatter plots of the number of literals x time of each run.</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):
Backtracking:
A dictionary/hash map to track assignments (True, False or None)
A set to track which variables have yet to be assigned
A stack to track tried literals and to backtrack if needed
Brute Force:
Cartesian product to produce all possible combinations of assignments
Best Case:
Cartesian product to produce all possible combinations of assignments</td></tr>
<tr><td>10</td><td>General operation of code (for each subproject):</td></tr>
</table>

	Backtracking: Backtracking uses a stack to track literal assignments. The algorithm will continue to assign literals as long as the problem is not unsatisfiable. If the existing assignments make it satisfiable, the program can terminate. If the most recent assignment makes it unsatisfiable, then the program starts to work backwards through previous assignments, trying out new assignments. In our program, literals are first tried as True; if that fails, then they are tried as False (a shallow backtrack). If False fails too, then that literal is popped, and a deep backtrack to the previous literal ensues. This backtracking goes on until a new literal assignment can be tried. If the stack becomes empty, then that means no assignment of literals could have worked, and the code terminates. The verifier for all these subproblems works to check if all clauses are true, if any are false, or if some are true and the rest are undeterminable. This uses simple negation and Boolean logic. Brute Force: Generate all possible combinations of assignments and test each one individually to see if it satisfies. If a solution is found, it returns it; otherwise, it will continue until it tests all combinations. If no solution is found, it returns false. Best Case: Generate all possible combinations of assignments and test each one individually to see if it satisfies. Also, has a counter that keeps track of the current maximum number of clauses satisfied. If a solution is found, it returns it; otherwise, it will continue until it tests all combinations and update the "number of clauses satisfied tracker". If no solution is found, it returns the best score or best case.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code: We did not add a testing suite, but we did work through samples for each subproblem to manually verify our algorithms – plus checked against data_kSAT_surgical.cnf, which contained the satisfiable and unsatisfiable attributes for us to check against. We wanted to use kSAT to check our code against varying literal counts, clause counts, and have the provided solutions to check against. This file showed us that our subproblem algorithms were working.
12	How you managed the code development: We forked GitHub, then everyone independently contributed their own subproblem algorithm. It was actually a very efficient and seamless development process – no merge issues or anything.
13	Detailed discussion of results: From our tests, our subproblems appear to perform correctly. In the plots, we can see the beginning of an exponential relationship between the number of literals and run time. This follows our expectations given that a kSAT program with k >= 3 is NP hard. Additionally, in our plots we can see that unsatisfiable problems tend to take far longer because they require the algorithm to fully exhaust options. Timing, we're pleased with each run batch taking a couple of seconds at most – we did try to overload with an input of 100 problems of varying k-sizes, and that took too long to run before we had to terminate the program. Indicating that k-SAT problems with large k's take substantially longer to run, as is the nature of NP-hard problems.
14	How team was organized: We all took one subproblem, and Sean Quigley also developed shared scripts – verifier function, graphic script – as well as orchestrated the GitHub / project setup.
15	What you might do differently if you did the project again: We are happy with our

	contributions, teamwork, and outcomes. For an exercise, though, we all think it would be valuable to implement some testing.
16	Any additional material: Utilized University of Washington's SAT Solver helper to verify our output: https://homes.cs.washington.edu/~kevinz/sat-solver/