# Project 1 Readme Team mfues

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: mfues</td></tr>
<tr><td>2</td><td>Team members names and netids: Maribella Fues, mfues</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: 2-SAT Solver using DPLL Algorithm</td></tr>
<tr><td>4</td><td>Overall success of the project: Successfully implemented 2-SAT Solver that accurately solves wffs in mostly-linear time</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 12</td></tr>
<tr><td>6</td><td>Link to github repository:<br>https://github.com/maribella-fues/theory_of_computing_project_1</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>2SAT_Solver_mfues.py</td><td>The 2-SAT Solver main code which includes an implemented DPLL algorithm. It can generate Satisfiability output, execution time trace output, or a scatter plot of time vs. number of variables.</td></tr>
<tr><td>2SAT_WFF_Generator_mfues.py</td><td>Code implemented to generate the test input file and data input files. "Randomly" creates a list of 2-SAT wffs. **Based off Professor Kogge's code in DumbSAT.py</td></tr>
<tr><td>DumbSAT_mfues.py</td><td>Altered DumbSAT code to generate test output to compare the 2-SAT Solver output with to determine correctness. **Based of Professor Kogge's code in DumbSAT.py</td></tr>
</table>
</td></tr>
</table>

| Test Files | |
|---|---|
| **check_2SAT_input_mfues.csv** | **Test input file to determine correctness of 2-SAT Solver execution. These wffs have known Satisfiability or Unsatisifiability.** |
| **check_2SAT_output_mfues.csv** | **Results file generated by DumbSAT using the same test input wffs to compare 2-SAT Solver output with and determine accuracy.** |
| **check_DumbSAT_2SAT_trace_mfues.csv** | **Trace file generated by DumbSAT using the same test input wffs to compare 2-SAT Solver trace with, particularly to compare execution times.** |
| **data_generated_small_2SAT_mfues.csv** | **List of wffs with up to 64 variables generated by 2SAT_WFF_Generator to build a scatter plot (takes about 3 minutes).** |
| **data_generated_large_2SAT_mfues.csv** | **List of wffs with up to 90 variables generated by 2SAT_WFF_Generator to build a scatter plot (takes about an hour).** |
| Output Files | |
| **output_2SAT_Solver_mfues.csv** | **Results file generated by the 2-SAT Solver from the test input file check_2SAT_input_mfues.csv.** |
| **output_2SAT_Solver_trace_mfues.csv** | **Trace file generated by the 2-SAT Solver from the test input file check_2SAT_input_mfues.csv.** |
| Plots (as needed) | |
| **plot_check_input_mfues.png** | **Scatter plot generated by the 2-SAT Solver using the input file check_2SAT_input_mfues.csv.** |
| **plot_data_small_mfues.png** | **Scatter plot generated by the 2-SAT Solver using the input file data_generated_small_2SAT_mfues.csv.** |
| **plot_data_large_mfues.png** | **Scatter plot generated by the 2-SAT Solver using the input file data_generated_large_2SAT_mfues.csv.** |

| | |
|---|---|
| 8 | Programming languages used, and associated libraries:<br>**Python with imported libraries of matplotlib.pyplot, time, copy, and random** |
| 9 | Key data structures (for each sub-project):<br>**2SAT_Solver_mfues.py:**<br>**Each wff is a list of lists with the internal lists representing each clause of literals. All wffs are stored in one large list of lists of lists to be iterated through to evaluate each wff. Each wff's number of variables & number of clauses are stored in their own lists, nvarlist and nclauselist respectively, so that the index of the wff in the list of wffs correlates to the indices of its corresponding number of variables & number of clauses in the other two lists. An Assignment list of lists is also created to be filled in for Satisfiable wffs using the DPLL Algorithm. The internal lists of Assignment correspond to each variable in the wff (variable 1 is at index 0 of the assignments list). The first element in the internal lists is the value assignment (0 for false, 1 for true) and the second indicates whether the variable has been assigned a value yet or not (0 for unassigned, 1 for assigned).**<br><br>**2SAT_WFF_Generator_mfues.py:**<br>**Uses a global list of lists called SAT2 to generate the wffs. Each internal list specifies, in order, the number of variables, number of clauses, number of literals, and number of wffs of that kind that the program should generate.**<br><br>**DumbSAT_mfues.py:**<br>**Uses the same SAT2 list of lists to generate the wffs. Also uses the same list of lists structure for each wff to evaluate. However, each wff is generated one at a time so they are not stored in a larger list structure like in the 2SAT_Solver. The Assignment list is a list of integers, unlike the 2SAT_Solver, with each integer corresponding to the value of the variable (0 for false, 1 for true). Once again, variable 1 is at index 0 of the Assignment list.** |
| 10 | General operation of code (for each subproject)<br>**2SAT_Solver_mfues.py:**<br>**The 2SAT Solver can generate three different types of output depending on which final function you run: test_execution() generates a basic results file that states the Satisfiability of each wff and its Assignment if it is Satisfiable, trace_execution() generates a trace file that includes execution time statistics for every 10 wffs run (based off Professor Kogge's DumbSAT code), and generate_scatter_plot() generates a execution time vs number of variables plot to visualize the relationship between the two. While each does something different with the information, all three functions call the build_wff() function and test_wff() function. The build_wff() function reads in the input file, assuming appropriate formatting, and builds three lists: the list of wffs and the lists of corresponding number of variables and number of clauses. All three final functions then iterate through the list of wffs, passing in each wff, number of variables, and number of clauses to test_wff() each time. test_wff() is based off of the same test_wff() function in Professor Kogge's DumbSAT code. It calls the DPLL() function (which actually solves the wff) and times how long the execution takes which it then returns. The DPLL() function is the actual 2-SAT solver algorithm that implements multiple methods to efficiently determine the wff's Satisfiability and assignments. It calls three main solving functions: unit_propagate(), remove_pure_literal(), and** |

| | |
|---|---|
| | **backtrack(). The functions findUnitClause() and pure_literal() are both helper functions for the first two algorithms to determine if they need to be called. set_assignment() is also called by the first two algorithms to edit the Assignments list as needed. backtrack() uses recursion which, more efficiently than DumbSAT, tests different assignments for variables that haven't been assigned yet to try to make the wff Satisfiable. backtrack() also calls unit_propagate() in its process to determine if the correct test assignment works. To run the 2SAT_Solver, you must decide which of the three final functions you want to run (test_execution(), trace_execition(), or generate_scatter_plot() ) depending on what output you would like from the solver. Each of the three functions take the input file as its argument to read the wffs in from.**<br><br>**2SAT_WFF_Generator.py:**<br>**The origin of this code is from Professor Kogge's DumbSAT code. It adapts the two functions build_wff() and generate_cases() to "randomly" generate wffs according to parameters specified in the data structure SAT2 and write them to a new input file to be used for the 2SAT_Solver. Each line written to the new input file must be in the format:**<br>**NumVars,NumClauses,[[wff clause1],[wff clause2],...]**<br><br>**DumbSAT_mfues.py:**<br>**The origin of this code is from Professor Kogge's DumbSAT code. This code was not altered much but it was used to generate the test output file to compare against the 2SAT_Solver output. While a very inefficient method of solving 2-SAT problems, the assignments are guaranteed to be correct and so it was used to generate a correct results file and a trace file to compare against the 2-SAT output. The main function is run_cases() which uses the 2SAT list of lists to "randomly generate" wffs using build_wff() to then solve. It calls the test_wff() function to time the execution of the solver and then the brute-force solving of the wff is in the check() function which implements many loops to test every possible assignment to determine Satisfiability.** |
| 11 | What test cases you used/added, why you used them, what did they tell you about the correctness of your code.<br>**check_2SAT_input_mfues.csv:**<br>**This file includes lines of randomly generated wffs produced by 2SAT_WFF_Generator.py. Because the code uses altered functions from the DumbSAT.py code and because random isn't actually purely "random," I used this data as test input for the 2SAT_Solver because I knew DumbSAT tests the exact same wffs and would generate results that are guaranteed to be correct (although inefficiently). Rather than having to create and solve a bunch of wffs myself to check that the 2SAT_Solver works correctly, I used this data because DumbSAT would generate the results for me and all I would have to do is compare if the 2SAT_Solver determined the same Satisfiability. The lines are formatted as: 'NumVars,NumClauses,[[wff clause1],[wff clause2],...]' because that is how the build_wff() function in the 2SAT_Solver expects to read in the data from each line of input.**<br><br>**check_2SAT_output_mfues.csv:**<br>**This file is the output generated by the DumbSAT.py solver using the same wff** |

| | |
|---|---|
| | **input as that from check_2SAT_input_mfues.csv. Although the DumbSAT.py code "randomly" generates the wffs, because random is not purely random, they are the same every time and therefore match the data input (I confirmed this when testing by print statements). This output can then be used to compare to the 2SAT_Solver output to make sure that it correctly determines Satisfiability.**<br><br>**check_DumbSAT_2SAT_trace_mfues.csv:**<br>**This file is the trace file generated by the DumbSAT.py solver. Once again, because "random" isn't purely random, the solver is guaranteed to be running the same wffs as those being inputted from the check_2SAT_input_mfues.csv file to the 2SAT_Solver (I also confirmed this by print statements). I wanted to use the trace file to compare against the 2SAT_Solver.py output because it would allow me to compare the execution time statistics between the two to see how much more efficiently the 2SAT_Solver.py determines the Satisfiability of the same data.**<br><br>**data_generated_small_2SAT_mfues.csv:**<br>**This data was generated using 2SAT_WFF_Generator.py to build a large number of wffs in order to generate enough points on the plot to be able to visually see the relationship between execution time and number of variables. The 2SAT wffs in it can have up to 64 variables. This data took the 2SAT_Solver about three minutes to generate a plot from. The larger number of wffs and variables better allowed the graph to show the linear trend.**<br><br>**data_generated_large_2SAT_mfues.csv:**<br>**This data was generated using 2SAT_WFF_Generator.py to build an even larger number of wffs in order to generate enough points on the plot to be able to visually see the results. The 2SAT wffs in it can have up to 90 variables. This data was so large that it took about an hour for the 2SAT_Solver to get through it and generate the plot. However, I found this very essential to my overall analysis because the 2SAT_Solver seemed to approach linear time the more variables it was given.** |
| 12 | How you managed the code development<br>**I drew a lot of inspiration from going over Professor Kogge's DumbSAT.py code file to figure out how to format and read in the wffs as well as how to get the execution time and format the output. I first spent a long time developing the DPLL algorithm to actually solve the 2SAT problems in an efficient manner. I approached each method one by one, first implementing unit propagation, then pure literal removal, and finally backtracking to finish the assignments. Once I had determined that this algorithm was running correctly, I then implemented the three final functions, depending on what output I wanted to get. I first implemented test_execution() to help me determine that the DPLL algorithm was running correctly because the output could be easily read & checked. I then implemented trace_execution() (based a lot off the trace file generated in DumbSAT.py) in order to compare the actual execution times of my solver to make sure it was significantly speeding up solving 2SAT problems (it was). Finally, I researched matplotlib and figured out how to use my generated results to make a scatter plot of execution time vs number of variables in order to evaluate my program's efficiency in a more visual manner.**<br>**In order to generate input and verify my results, I spent a lot of time manipulating** |

| | |
|---|---|
| | **DumbSAT.py and its internal functions to produce the input and output files that I wanted.** |
| 13 | Detailed discussion of results:<br>**To verify the correctness of my algorithm, I used output_2SAT_Solver_mfues.csv. This is the results file generated by running test_execution() in the 2SAT_Solver.py. I verified the correctness of these results by comparing them to check_2SAT_output_mfues.csv, which is the correct expected output for the data inputted. By nature of solving wffs, there can at times be many different assignments that satisfy the wff. This is the reason why the assignment output in output_2SAT_Solver_mfues.csv is different from check_2SAT_output_mfues.csv because the wff was solved in a different manner. However, I verified the first few that differed to make sure that the assignments generated by the 2SAT_Solver still satisfy each wff, which it does. More importantly, the two files are identical in determining whether or not each wff is Satisfiable or Unsatisfiable which I used to confirm that my 2SAT_Solver was indeed working correctly.**<br><br>**Next, I wanted to verify that my 2SAT_Solver is solving the wffs in a more time-efficient manner than the DumbSAT Solver. I did this by generating a trace output from the 2SAT_Solver using trace_execution(), which is stored in the file output_2SAT_Solver_trace_mfues.csv. I then compared this output to the file check_DumbSAT_2SAT_trace_mfues.csv which was the trace file generated by Professor Kogge's DumbSAT algorithm on the same wffs. Comparing the two files, I noticed how the DPLL algorithm in the 2SAT_Solver solves the same number of wffs significantly faster than the DumbSAT Solver. Interestingly, the 2SAT_Solver takes longer to solve smaller wffs with a smaller number of variables - the DumbSAT Solver actually, on average, solved both Satisfiable and Unsatisfiable 4 variable and 8 variable wffs faster than the 2SAT_Solver. However, the breaking point was reached once the wffs contained 12 variables. After that point, the 2SAT_Solver ran significantly faster than the DumbSAT to solve both Satisfiable and Unsatisfiable wffs. Therefore, I deemed the 2SAT_Solver to be a success at solving wffs in a remarkably more efficient manner than the DumbSAT 2SAT Solver.**<br><br>**Finally, to continue verifying the effects of the 2SAT_Solver on execution time, I generated multiple scatter plots to visualize the execution time vs. the number of variables. The first plot, plot_check_input_mfues.png, was generated using the wffs from the input file check_2SAT_input_mfues.csv. This file doesn't go super high in the number of variables so it wasn't the best or largest data set to check. Looking at the different points, an exponential curve is quite obvious. However, this curve is formed by only a few of the wffs (mostly Unsatisfiable ones), with the majority of the points being quite close to the axis which reflects that the 2SAT_Solver was able to solve most of the wffs in a much faster manner.**<br><br>**The next plot, plot_data_small_mfues.png, was generated using the wffs from the data file data_generated_small_mfues.csv, which included wffs with a much larger range of variables (up to 64 variables). This plot was a lot more reflective of how implementing the DPLL algorithm in the 2SAT_Solver moves it closer to solving the wffs in linear time rather than exponential. While there are few outlier Unsatisfiable wffs that do form an exponential curve, this is about 10 of 280 wffs** |

| | |
|---|---|
| | **represented on the graph. Ignoring the outliers, the large majority of the wffs are still very close to the x-axis, with the top of the scatter forming a slightly-increasing line, which is not exponential growth.**<br><br>**The larger data set was able to show much better the significant impact that the 2SAT_Solver had on the execution time and so I wanted to see what an even larger data set would generate. The third plot, plot_data_large_mfues.png, was generated using the data file data_generated_large_mfues.csv, which included wffs with up to 90 variables. This plot took a LOT longer to generate (about an hour) but it shows how the execution time for the 2SAT_Solver is not exponential. There are about 9 outliers (four of which are definitely the main reason it took so long) but the rest of the 341 wffs in the input file are all concentrated around the x-axis and slightly increase linearly. The graph, while a monster to generate, clearly demonstrates how the 2SAT_Solver is able to solve the wffs in non-exponential time.** |
| 14 | How team was organized<br>**N/A, I did all the tasks.** |
| 15 | What you might do differently if you did the project again<br>**If I could start the project differently, I would determine what I wanted to have as my output/results before I started. When I first started, I had envisioned that most of my time spent on the project would be attempting to program the actual algorithm to solve the 2SAT wffs. In reality, the majority of my time went into figuring out how to generate input with known output and how I would verify that the solver itself was correct. If I had gone in with a vision for how I wanted to verify the results, instead of figuring out in the moment how to prove it, it would have saved me time testing different things to figure out how I wanted to relay and check the results.**<br>**I also would find a different final method in the DPLL algorithm rather than backtracking, because recursion can use a lot of memory space. For more realistic SAT problems that use millions of variables, the amount of memory space used by the recursive function would be enormous, which is not ideal.** |
| 16 | Any additional material: |