Project 2 Readme Team msulli52

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"

1	Team Name: **msulli52**
2	Team members names and netids: **Molly Sullivan, msulli52**
3	Overall project attempted, with sub-projects: **The "Knapsack" Problem: An equivalent of DumbSAT for a solver that returns a coin combination that works**
4	Overall success of the project: **My project was successful.**
5	Approximately total time (in hours) to complete: **12 hours**
6	Link to github repository: https://github.com/msulli52/theory_project2_msulli52/
7	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.

File/folder Name	File Contents and Use
Code Files	
traceTM_msulli52.py	**This python file holds all the algorithmic code for tracing all possible paths an NTM might take to traverse a string to determine if it is accepted or rejected by a defined input language. It also does all the complementary work of initializing the NTM, parsing the input machine file, running the bfs algorithm on a tree structure, calculating nondeterminism, and generating output in a table. It takes in all the test files listed below as input, and it generates output stored in the output files listed below as well.**
Test Files	
data_a_plus_msulli52.csv **data_a_plus_DTM_msulli52.csv** **data_abc_star_msulli52.csv**	**All of these files define a Turing machine for recognizing a certain language. Their contents include the name of the**

data_abc_star_DTM_msulli52.csv data_equal_01s_msulli52.csv data_equal_01s_DTM_msulli52.csv data_even_1s_msulli52.csv data_even_1s_DTM_msulli52.csv	**machine, list of states, list of input symbols, list of tape symbols, start state, accept state, reject state, and all transition rules for the machine and its language. The files are NTMs, except for those which have DTM specified in their name. These should all be in the same directory as traceTM_msulli52.py as they are called in that python code as test case input.**
Output Files	
output_msulli52.txt output_table_msulli52.txt	**I have two different output files that present the same information in different ways. The first, output_msulli52.txt, lists the output results for each test machine and string (machine name, initial string, total transitions, nondeterminism, and traversed path) one by one in a text file. The second, output_table_msulli52.txt, presents the same information in a table format, each output field having its own column for side by side easier comparison.**
Other	
readme_even_1s_msulli52.txt	**I created this readme file to explain the two input data files I created for extra credit, data_even_1s_msulli52.csv and data_even_1s_DTM_msulli52.csv. I thought I would include it in the project submission as well. It outlines how/why I created it, the file contents, how to use the files and examples of expected behavior on different test strings. All the other input data files I got from the shared drive.**

8	Programming languages used, and associated libraries: **Python, csv, tabulate**
9	Key data structures (for each sub-project): ● **Self.transitions: dictionary** ○ **This dictionary stores all the transitions of the Turing machine** ○ **Keys: a tuple (state, input_symbol)** ○ **Values: a list of tuples (next_state, write_symbol, move_direction)**

	● **Tree: list of lists** ○ **Used a list of lists to represent the path of configurations that the TM follows as it traverses a string** ○ **Each level is a list of tuples (left, state, right) to keep track of the current state of the machine as well as the tape contents to the left and right** ● **Visited: set** ○ **This set tracks the visited configurations to prevent infinite loops caused by revisiting the same configuration** ○ **Holds a tuple (left, state, right) to represent the configuration** ● **Next_level: list** ○ **This list holds the configurations generated from the current level of the tree to be processed in the next step** ○ **The list is made up of tuple configurations (left, state, right)** ● **Current_level: list** ○ **This list represents the current level of configurations in the computation tree being processed**
10	General operation of code (for each subproject): **My program, traceTM_msulli52.py, simulates the behavior of both deterministic and non-deterministic Turing machines. At a high level, it traces all possible paths that an NTM/DTM might take when handling an input string on a machine for a specific defined language. It uses a breadth first search (BFS) approach to the traversal and uses a list of lists as the main data structure to represent the TM tree traversal of various paths and tape configurations. After executing the simulation on a variety of input machines and strings, it produces an output table for comparison of machine name, initial string, total transitions, nondeterminism, and traversed path. I use an NTM class to store transitions, machine name, start state, accept state, and reject state.** **The execution begins with the parse_machine_file method, which reads the input CSV file describing the Turing machine. This file specifies the machine's states, alphabet, start, accept, and reject states, as well as its transition rules. The transitions are stored in a dictionary (self.transitions) for efficient lookup during the simulation.** **The run method executes the entire simulation process. It maintains a tree-like structure (tree) to represent the computation paths of the machine. At each step, the program processes configurations from the current level of the tree (current_level), determines valid transitions, and generates the next level of configurations (next_level), a breadth- first search approach. A set (visited) ensures that repeated configurations are not revisited, preventing infinite loops. The program halts if an accept state is reached, all paths lead to rejection, or the specified depth limit is exceeded. In case of a depth limit, the execution is marked as "stopped due to depth limit."** **The calculate_nondeterminism method analyzes the simulation to compute the nondeterminism, or average number of transitions per configuration, offering insights into the machine's complexity. The results of the simulation, including**

	the machine's name, input string, depth of the tree, total transitions, nondeterminism, and execution path, are formatted and summarized using the generate_output method. **In the main section, multiple input files and test strings are iteratively processed. Each machine's results are written to both output_msulli52.txt and output_table_msulli52.txt for both a list and table view of all the outputs, respectively.**
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code: **I have 8 different data input files, each with 4 different test strings, totaling to 32 test cases for my code. Of the 8 data files, 4 are NTMs and 4 are DTMs. I chose four different languages to use, one NTM and one DTM for each: a plus, abc star, equal number of 0s and 1s, and even number of 1s. I created both of the data input files for the even number of 1s language to get practice setting up my own input file and language definition, and the other three are from the shared drive. Testing out four different strings for each language, provided a wide variety of test cases to prove that my implementation worked properly. I made sure to include strings of all sizes, both accept and reject strings, the empty string, strings of length 1, strings that were so long they reached the max depth limit, etc. I checked over each test case and even used more cases through the code development process than I kept for submission, to ensure I was getting the expected output for each string. I did some hand calculations and traversals of parse trees as well to verify my work. Having a clear trace of the bfs parse tree in my output was helpful in confirming correctness. Overall, using a wide variety of test cases, I can verify that my traceTM algorithm works correctly for both NTMs and DTMs.**
12	How you managed the code development: **Since the assignment description document suggested using a bfs approach to traversal while using a tree (list of list) data structure, I used that as my main strategy throughout. I made sure to go through the assignment document really closely before starting and took notes on how to manage input and output to make sure I was doing everything diligently and correctly from the start. I decided to use a class structure to keep track of all the pieces of the NTM definition. Then from there I figured out what key data structures I would use to store information and connect everything. Then I used my previous knowledge on BFS to build an algorithm that would traverse a variety of input files. I tested my code as I went with all my input machines and strings, modifying my code if I saw issues arise in my output. Sometimes the algorithm would work correctly for some machines, but not others, so I had to make adjustments. That's where it was helpful to have a wide variety of test cases to ensure validity. At the end I turned my output_msulli52.txt running output into a table (output_table_msulli52.txt) for easier viewing and comparison using tabulate. I also created my own test files in addition to the given ones to get experience with creating my own for more understanding and extra credit. I did all the code locally on my computer on VS Code and then pushed all the files to GitHub for submission.**

13	Detailed discussion of results: **The results of my program provide a comprehensive trace of the computation for various NTMs and DTMs. Each machine was tested on multiple input strings, resulting in a database of the tree depth, number of transitions, nondeterminism, and traversal path of several machines. These results demonstrate the correctness, efficiency, and flexibility of the code, as well as highlight the differences in behavior between deterministic and non-deterministic machines. There were a few patterns that I noticed in the results. Nondeterminism increases with TMs that have more non-deterministic transitions, meaning they have more branch/transition options at each level to explore before continuing down. This is especially evident in abc star, which has a much higher nondeterminism than the other languages, ranging from 2-4ish while others staying more around 1-2. It's a slightly more complicated language to traverse, opening opportunities for more branching when traversing. The opposite is seen in the even number of 1s language, which has a nondeterminism of 1 for both the NTM and DTM, because the machine's transition rules lead to only one valid configuration at each step. Also, longer strings generally have a higher degree of nondeterminism because there are more opportunities for branching. This can be seen in the strings aaa (1.75) vs aaaaaaaaaaaaa (1.93) in the a plus language, among others. DTMs always have a nondeterminism of 1, as my results show, because they only ever have one transition for any given state, there is no branching. My results also include edge cases, such as the empty string, invalid strings, and strings that reach the depth limit.**
14	How team was organized: **I worked by myself on this project.**
15	What you might do differently if you did the project again: **I would have spent more time testing more iteratively. I spent a lot of hours debugging my program until I could finally get it working correctly on several different machines. I wrote all the code and then tested it on all 4 machines, so debugging the entire program was probably more time consuming than testing bit by bit and really printing out each part of the traversal process as I went. I think I could have definitely saved myself a few hours of being stuck.**
16	Any additional material: **Just the extra credit tester files and readme that are in the github already:** **data_even_1s_msulli52.csv** **data_even_1s_DTM_msulli52.csv** **readme_even_1s_msulli52.txt**