Project 2 Readme Team alyssariter

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: alyssariter</td></tr>
<tr><td>2</td><td>Team members names and netids: Alyssa Riter - ariter</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: Program 1: Tracing NTM Behavior</td></tr>
<tr><td>4</td><td>Overall success of the project: Overall, this project is successful in parsing a NTM based on the definition of the machine given in the CSV test files.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 7 hours</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/AlyssaRiter23/Tracing-NTM-Behavior</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>Code Files/traceTM_alyssariter.py</td><td>This is the code file that conducts a breadth first search through a NTM given by a user inputted csv file. This program creates the configurations, explores the various branches of a configuration tree, prints the tree created as well as the machine name, input string, number of transitions, depth of the tree and whether or not the string was accepted or rejected. It prints both to the terminal and to an output file.</td></tr>
<tr><td colspan="2">Test Files</td></tr>
<tr><td>Test Files/test.csv
Test Files/test1.csv
Test Files/test2.csv
Test Files/test3.csv
Test Files/test4.csv</td><td>These test files are used to test whether or not the BFS works. Each test file is organized as such:
Line 1: machine name
Line 2: Q, the set of finite states
Line 3: Σ, the input alphabet
Line 4: Γ, the tape alphabet</td></tr>
</table>
</td></tr>
</table>

		Line 5: the start state **Line 6: F, the set of accept states** **Line 7: the reject state** **All further lines are the transitions of the machine** **Note test1.csv and test2.csv are DTM while the rest are NTM and produce full trees for the output.**
	Output Files	
	Output Files/output.txt	**This file holds the output of each test run on the program. After traceTM_ariter.py is run the result is appended to this file. For each run this output file shows:** **1. The Input String:** **2. Machine Name:** **3. String accepted in {transition_count} transitions** **4. Depth of the Tree:** **5. Number of Configurations** **6. Average Non-determinism** **7. The Explored Paths Tree (which shows all possible paths to explore with the NTM) and ends in an accept state if the string can be accepted.**
	Plots (as needed)	
	table_NTM_alyssariter.csv **plot.py**	**This file holds a table of the output of each test run on the program. For each run this output file shows:** **1. The Input String** **2. Machine Name** **3. Whether or not the string was accepted** **4. Number of transitions** **5. Depth of the Tree** **6. Number of Configurations** **7. Average Non-determinism** **plot.py is just the script used to turn the output file of the program (output.txt) into a csv file**
8	Programming languages used, and associated libraries: **python, python's collections module to use a deque and argparse for parsing command line arguments. Also used regular expressions for extracting important information for creating a table from the output file. I also used pandas for outputting results as a csv file.**	

9	Key data structures (for each sub-project): **For Tracing NTM Behavior: tree for listing the configurations and explored paths, a deque for double ended access to the nodes of the tree, a list for storing the information about each configuration (current state and tape), and a dictionary to store information about the machine (machine name, start state, accept and reject states, set of states, alphabet, tape alphabet, transitions)**
10	General operation of code (for each subproject): **The program begins by parsing the command-line options provided when the program is run. The available flags include:** ● **-f to specify the machine definition from a CSV file.** ● **-i to specify the input string.** ● **-d to set a maximum depth.** ● **-t to limit the maximum number of transitions.** ● **--debug to enable debugging print statements.** **The specified file is processed by the `parse_file` function, while the other arguments are passed to the `turing_machine_bfs` function.** **The `parse_file` function opens the file for reading and initializes the machine's characteristics, assuming the format is valid. The characteristics include: the machine name, Q - the set of states), Σ - the input alphabet, Γ - the tape alphabet, the start state, F - the set of accept states, the reject state, and the transitions. These characteristics are returned as a dictionary, mimicking the traditional Turing Machine tuple definition.** **The `turing_machine_bfs` function starts by opening an output.txt file to append results. If the input string is empty, a blank symbol is concatenated to ensure proper processing. The function initializes the transitions (that were defined by `parse_file`), a list to track explored paths, and counters for transitions and configurations. The root node is added to a queue, and then we begin the breadth-first search using a `while` loop. While the queue is not empty, nodes are dequeued, and the new configuration's attributes (state, tape, head position, and depth) are extracted. The explored paths tree is updated by storing the part of the tape before the head, the state we are in, and then the part of the tape at and after the head, in a tree-like structure, emulating the BFS style exploration of each depth before descending further.** **Within the loop, the program checks if the current state is an accept state. If so, it prints the results (machine name, input string, transitions, configurations, tree depth, and explored paths tree). It also checks if the maximum transitions or depth limits have been exceeded, printing the results in those cases as well if they are exceeded. If any of these conditions are met the search ends. The use of a maximum depth and maximum transitions prevents the search/machine from looping infinitely if a string is not accepted.** **For each valid transition, the program creates new configurations by copying the tape and applying the transition rules to update the head symbol and position. If the head moves out of bounds (e.g., negative positions), a blank symbol is added,**

	and the head is reset. New child configurations are created and added to the queue if the string remains unaccepted so that we continue searching for a path to acceptance. If the queue is exhausted without accepting the input, the program prints the final simulation results. **To manage configurations, a `configuration` class is used. Each node in the tree represents a configuration and includes attributes for state, tape, head position, depth, and parent node (with the root node's parent set to `None`).** **For `plot.py` I read from the output.txt file and use regular expressions to extract the information needed to create a table. This script is fairly straightforward once you understand what regex you need for each element to be displayed in the table. After extracting the information I ensure that there is a value for each input string and display none if no information for a certain header is extracted. I append all the information for each string to a list and then use python's pandas to import it to a CSV file.**
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code: **The test cases I used were the NTM a+, the deterministic turing machine (DTM) for the language 0^2n where n >= 0, the DTM to represent the language {w#w | w is an combination of 1s and 0s}, the NTM for the language accepting strings represented by the regex a*b*c*, and the NTM accepting the language where strings have an equal number of 1s and 0s. I used these test cases to ensure that my program could handle cases where the machines were deterministic and non-deterministic and to ensure that deterministic machines only had one path to either accept or reject and that the non-deterministic machine had a tree of multiple paths where some paths did not lead to an accept state but others did. This ensured that my breadth-first search was working and accurate. Additionally, I used numerous machines to test my program to ensure that it was robust and could work with a variety of transitions, alphabets, and configurations. Since these NTMs and DTMs were based on class examples and textbook problems they were relatively easy to trace by hand in order to ensure that the tree outputted by my program was correct and represented all possible paths that the input string could lead to. I was able to count the transitions by hand as well to verify that this counter was accurate. Overall, since the output of my code matched the results of calculations done in class and by hand I can assert that my program is accurate.**
12	How you managed the code development: **I developed all the code in one python program with 3 different functions: one for parsing the CSV file, one for conducting the breadth-first search on the machine and input string, and one for printing out the tree that was explored during the breadth-first search. I also used test files that were formatted as CSV files because they were helpful in ensuring the correctness of the code since they were based off of problems we did in class and that are found in the textbook. All of my code was organized and stored in my Github repository to allow for efficient organization and access to my code history.**
13	Detailed discussion of results: **The machine generates multiple nodes at each level**

	of the tree during a breadth-first search on non-deterministic Turing Machines (NTMs). For example, the machines for a+, a*b*c*, and {w | w has an equal number of 1s and 0s} produce trees that have various paths explored for each input string. This behavior aligns with the nature of an NTM, since it is a theoretical model that can have more than one possible action when in a given state with a given input. As is often seen in the output of my program, the total number of transitions is greater than the depth of the tree for a NTM, whereas for a DTM, the number of transitions is equal to the tree's depth. An input to an NTM is accepted if at least one node in the tree represents an accept configuration; otherwise, it is rejected. From the program we observe that a deterministic machine has an average non-determinism of ~1 - this means that the machine has only one viable choice at each step or for each input, with no branching or exploration of multiple paths. In contrast, the machines that have a high average non-determinism make many non-deterministic choices at each step and explore multiple paths or options simultaneously; this can be seen in the NTMs that have many nodes at each level of the tree. If the non-determinism is very high, the number of possible computational paths can grow exponentially with the input size, which is seen in NP-complete problems as observed in project 1. To prevent our NTMs from looping on long inputs, I included a maximum depth and maximum number of transitions to halt execution if we witnessed problems becoming too large.
14	How the team was organized: **It was a single person team so I created the NTM tracing program, test files, and output files.**
15	What you might do differently if you did the project again: **I would change the method of printing, as it is redundant to have three different sets of print statements and it would have been much easier and more efficient to just make the printing occur in a separate function.**
16	Any additional material: **N/A**