Project 2 Readme Team MJ

12/08/24

1	Team Name: MJ
2	Team members names and netids: Max Johnson, mjohns79
3	Overall project attempted, with sub-projects: Program 1, Tracing NTM behavior
4	Overall success of the project: GOOD! Since I was working by myself, it was easy to get everything done. Outputs all worked and I enjoyed implementing a NTM.
5	Approximately total time (in hours) to complete: 7 hrs
6	Link to github repository: https://github.com/maxj723/theory-project-two
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/folder contents and use**
Code Files	
traceTM_MJ.py	It contains all the code to build and run an NTM. Given a CSV, input string, and max depth, it will print the desired output.
Test Files	
check_tests_MJ.py	Test file to check code output for traceTM_MJ.py run it using **python3 check_tests_MJ.py**
check_MJ folder	Contains all NTM csvs for testing purposes
a_plus_DTM.csv	csv for DTM of aplus language
a_plus.csv	csv for NTM of aplus language
abc_star_DTM.csv	csv for DTM of a*b*c* language
abc_star.csv	csv for NTM of a*b*c* language
equal_01s_DTM.csv	csv for DTM of {w | w has the same number of 0's and 1's}
equal_01s.csv	csv for NTM of {w | w has the same number of 0's and 1's}
Output Files	
output_MJ folder	Contains all screenshots of testing outputs

output_abcstar_MJ.png	screenshot of output from running different inputs on abcstar NTM
output_abcstarDTM_MJ.png	screenshot of output from running different inputs on abcstar DTM
output_aplus_MJ.png	screenshot of output from running different inputs on aplus NTM and DTM
output_equals01s_DTM(1)_MJ.png	screenshot of output from running different inputs on equals01s DTM
output_equals01s_DTM(2)_MJ.png	more screenshots of output from running different inputs on equals01s DTM
output_equals01s(1)_MJ.png	screenshot of output from running different inputs on equals01s NTM
output_equals01s(2)_MJ.png	more screenshots of output from running different inputs on equals01s NTM
output_test_MJ.png	screenshot of output from check_tests_MJ.py. This just shows that the test works and outputs are correct

8	Programming languages used, and associated libraries: Python – csv, argparse
9	Key data structures (for each sub-project): NTM class (self made), Tree (to hold possible configs)
10	General operation of code (for each subproject): For traceTM_MJ.py, I built an NTM class that contains the attributes and functionality of a non-deterministic turing machine. The class works as follows: **__init__() function:** sets all values of the formal definition tuple given inputs to the attributes of the NTM. **get_next_config() function:** given a config, it determines all possible next configurations using the transition function attribute of the NTM. This is first done by checking the current state and head symbol and identifying if it is possible to transition. Then, it will complete all transitions available and output a new configuration for each. The function returns a list of all the possible configs given. If there are none, it will output a reject config. **process_input() function:** takes an input string and max_depth parameters and runs the turing machine on them. It will print results. The function starts by initializing a tree with the start state as the root config. Then, BFS is used to search the tree for an accepted configuration. When at a certain level, it iterates through each config in that

	level and checks for accept/reject, then finds all the next possible configs after that state using the get_next_config(). Once it has iterated through all the configs in the layer, it adds the next layer configs to the next layer in the tree until an exit is hit. Then, the output is printed. **print_output() function:** prints the output in the desired way as described by the directions. The main script has a process_csv() function that utilizes the csv library to read the input CSV file and convert it to an NTM object. This NTM object is returned. The main function uses the argparse library to read in cmd line arguments and use the process_csv() function to run the NTM's process_input(). This will result in the desired output. Example usage: **python3 traceTM_MJ.py Test_files/a_plus.csv aaa 7** The first argument after the file name is the NTM CSV file, the second argument is the input string, and the third argument is the max depth.

11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. I made 5 different test cases to test my code. First, I used the aplus TM to test because it was provided in the writeup, so I knew that it would be correct for the tests. TEST1 used a short input that would accept but also made it very easy to follow the tree to confirm it was correct. TEST2 used an incorrect character. This would prove that machine would correctly reject TEST3 processed a longer string but with a small max depth. This was to test how the max_depth_exceeded feature would work, and how the output printed as well. TEST4 processed a much longer string just to see how it would fair with longer inputs as well as proper output printing. TEST5 tested a longer input with a bad character. This was to make sure it would still reject. These tests were just to confirm the code worked. I ran the program on other TMs to identify how nondeterminism changes with machines.
12	How you managed the code development Since I was working alone, it was easier to manage. I developed one function at a time and tried to test it individually. This made it easier to test at the end because I knew each function worked properly as I went. This made it easier to identify what could be causing the problems and how I could fix them. I would have liked to commit my work more often to prevent the possibility of losing saves, though. Using git was convenient for keeping all of the code and tests together.

13 Detailed discussion of results:
Table with results for NTM:

NTM	String	Result	Depth	#configs	Avg ND
a_plus.csv	aaa	accept	4	10	2.5
a_plus_DTM.csv	aaa	accept	4	4	1
a_plus.csv	b	reject	1	1	1
abc_star.csv	aabbcc	accept	7	47	6.71
abc_star.csv	cc	accept	3	7	2.33
abc_star.csv	aba	reject	3	18	6
abc_star_DTM.csv	cc	accept	3	3	1
abc_star_DTM.csv	aabbcc	accept	7	7	1
equal_01s.csv	0011	accept	15	22	1.47
equal_01s.csv	01010101	accept	41	65	1.59
equal_01s.csv	010	reject	6	9	1.5
equal_01s_DTM.csv	0011	accept	15	15	1
equal_01s_DTM.csv	01010101	accept	41	41	1

Given the three different TMs, it is clear that the DTMs always had the same number of configs as depth. This makes sense because they are deterministic, and the accept/reject path will always follow one path. The NTMs showed different results. a*b*c* had a very high level of nondeterminism. This could be because of the way * works, where it can include 0 occurrences or as many as wanted. equal_10s had a much lower nondeterminism at around 1.5 for each test.
The average nondeterminism stat is interesting because regardless of how long the input or how deep the search went, the average nondeterminism for NTMs stayed relatively the same. This shows that the results depend much more heavily on the language and the machine rather than the input or input length

14 How team was organized: Team only consisted of me, so I did all the work

15 What you might do differently if you did the project again: I would probably work with a partner so that I would have more time to explore the deeper options and have fun experimenting with other ideas.

16 Any additional material: