Project 1 Readme Team ekoran

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**: ekoran
2	**Team members names and netids** Ethan Koran NetID: ekoran
3	**Overall project attempted, with sub-projects:** Tracing NTM Behavior
4	**Overall success of the project**: My project was successful
5	**Approximately total time (in hours) to complete**: ~10 hours
6	**Link to github repository**: https://github.com/ethankoran/TOC_project02_ekoran.git
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.

<table>
<tr><td></td><td>
<table>
<tr><th>File/folder Name</th></tr>
<tr><td>Code Files</td></tr>
<tr><td>traceNTM_ekoran.py → code that traces the behavior of a turing machine file running on a given input string</td></tr>
<tr><td>Test Files</td></tr>
<tr><td>check_a_plus_DTM_ekoran.csv → deterministic turing machine that accepts the strings satisfying the regex a+
check_a_plus_ekoran.csv → non-deterministic version of above machine
check_abc_star_DTM_ekoran.csv → deterministic turing machine that accepts strings satisfying the regex a*b*c*
check_abc_star_ekoran.csv → non-deterministic version of above machine
check_equals_01s_DTM_ekoran.csv → deterministic turing machine that accepts strings with an equal amount of 1s and 0s
check_equals_01s_ekoran.csv → non-deterministic version of above machine
check_abPalindrome_DTM_ekoran.csv → deterministic turing machine that accepts strings that are palindromes (spelled the same way forwards and backwards)</td></tr>
<tr><td>Output Files</td></tr>
<tr><td>output_a_plus_NTM_ekoran.pdf → output file that contains both accept and reject cases for the check_a_plus_ekoran.csv test file
output_abc_star_NTM_ekoran.pdf → output file that contains both accept and reject cases for the check_abc_star_ekoran.csv test file
output_abPalindrome_DTM_ekoran.pdf → output file that contains both accept and reject cases for the check_abPalindrome_DTM_ekoran.csv test file
output_equals_01s_NTM_ekoran.pdf → output file that contains both accept and reject cases for the check_equals_01s_ekoran.csv test file</td></tr>
<tr><td>Plots (as needed)</td></tr>
<tr><td>output_NTM_table_ekoran → a table that includes a summary of the above output files</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries: I used Python to write the NTM tracer script, utilizing the sys library as well as file manipulation</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):
- TURING class structure to store general information about the turing machine, as well as a helper function to print out this information
- frontier, used as a queue, to implement breadth-first search
- A tree, which was a list of lists of lists that stores the possible configurations of the string at each transition level of the NTM</td></tr>
<tr><td>10</td><td>General operation of code (for each subproject)
The code first reads the TM's specification from a file, which includes states, transitions, and other data, and stores it in a data class. Then, taking the input string provided by</td></tr>
</table>

	the user, the program enters the turing_machine_BFS function, which traces the ™'s operation by simulating each transition. Each transition level is added to a tree structure, which contains all possible configurations at that level. This is done using breadth-first search, as this allows us to explore each branch in parallel. Once an accept state is reached, it back tracks through the tree to find the accepting path (the path that the machine took to get to the accept state). If the machine is deterministic, this can be done by just printing the levels of the tree. For nondeterministic machines, you must start from the last level (accept), repeatedly find the configuration in the level above that got you to that level. With each accept, the machine name, number of transitions explored, depth of the tree, accepted path, and the average nondeterminism are printed. If the machine never reaches an accept, or if all branches lead to a reject state, we merely print the number of steps it took to reject the string. Finally, if the number of transition levels exceeds the user's set limit, the execution stops.
11	**What test cases you used/added, why you used them, what did they tell you about the correctness of your code**. I used a deterministic and non-deterministic version of three different turing machines to test my code: one that accepts the regex a+, one that accepts the regex a*b*c*, and one that accepts palindromes. I used a variety of different test cases to ensure overall correctness of multiple inputs, and to ensure that my code was not written to only work on a certain machine. My code was able to produce correct results on all three machines, which I verified by writing out the accepting path by hand.
12	**How you managed the code development** First, I brainstormed the steps required to create my NTM tracer. To do this, I utilized resources such as a white board to visualize my strategy. I broke the process into three major steps: 1. Writing and debugging code for taking in test data input 2. Brainstorming data structures needed 3. Writing my BFS function to trace the behavior of the NTM transitions 4. Writing the back tracking function to find accepting path 5. Formatting output Breaking up my development process like this helped keep it more manageable as I worked
13	**Detailed discussion of results**: My code worked very well. After running multiple tests on multiple turing machines, I was able to create the following table:

NTM	Input String	Result	# Depth of Tree	# Configurations Explored	# Average Nondeterminism
check_a_plus_ekoran.csv	aaaa	Accept	5	12	2.167
check_a_plus_ekoran.csv	aaaaaaaaaa	Accept	12	33	2.615
check_a_plus_ekoran.csv	empty string	Reject	1	1	1
check_abc_star_ekoran.csv	aaabbbccc	Accept	10	43	4
check_abc_star_ekoran.csv	aaabbc	Accept	7	7	1
check_abc_star_ekoran.csv	bbbc	Accept	5	5	1
check_abc_star_ekoran.csv	abca	Reject	4	4	1
check_equal_01s_ekoran.csv	"0101"	Accept	13	20	1.5
check_equal_01s_ekoran.csv	"0011"	Accept	15	21	1.375
check_equal_01s_ekoran.csv	"00011"	Reject	20	23	1.15
check_equal_01s_ekoran.csv	"101"	Reject	6	8	1.333

From this table, there are multiple takeaways:

1. The average non-determinism, which is the number of total transitions explored divided by the depth of the tree, for non-deterministic machines is often greater than 1, as each state configuration can have multiple valid transitions out of them (hence non-deterministic). However, DTM's always have a degree of non-determinism of 1, since each configuration will only ever have one valid transition out of them
2. Generally, the longer the input string for each machine, the greater the depth of the tree and the greater number of configurations explored
3. Also, many of my reject cases had low average nondeterminism, possibly because they were rejected early.

I also added additional functionality at the end of each NTM trace that prints a table showing the number of transitions, branches, and nonleaves at each level of the trace. This was interesting to see how many branches ended at each level, and how many continued to produce longer output.

14 **How team was organized**:
I worked on this project alone, so I did each aspect of the project, starting with brainstorming, then the actual code, and finally the

15 **What you might do differently if you did the project again**:
If I were to do this project again, I would try to write my own test cases to further explore the results of my code. I would also analyze the affect of different length input strings. Although, I still achieved great results with the test cases that were provided to me.

16 **Any additional material**: N/A