Project 2 Readme Team Sajith

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: Sajith</td></tr>
<tr><td>2</td><td>Team members names and netids: Sajith Devareddy (sdevared)</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: Project 1: Tracing NTM Behavior</td></tr>
<tr><td>4</td><td>Overall success of the project: Successful</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 13</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/SajithDev/Tracing-NTM_TOC_Sajith</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>traceTM_Sajith.py</td><td>This is the main program that traces a TM. It takes in the input of the txt file of test cases and parses each case and uses the TMs provided in the input directory. It performs the tracing and outputs all relevant information as needed.</td></tr>
<tr><td colspan="2">Test Files</td></tr>
<tr><td>data (folder)</td><td>This contains the 6 TMs used in the testing process as explained more in detailed below (in code generation explanation)</td></tr>
<tr><td>data_test_cases_Sajith.txt</td><td>This is the txt file that contains the 14 test cases run using the TM definitions inside the data folder. Each TM has 2 test cases, except one having 4 to show unique tracing cases</td></tr>
<tr><td colspan="2">Output Files</td></tr>
<tr><td>outputs (folder)</td><td>This folder contains the outputs for each of the 14 test</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td></td><td>cases, the outputs are named in the following format:

machine_{TM number}_test_{test case number for that machine}</td></tr>
<tr><td colspan="2">NTM Tracing Table</td></tr>
<tr><td>output_table_Sajith.csv</td><td>This is a CSV that is visualized as a table in Github that shows the following as needed in the requirements for this project.

1. NTM used
2. String used
3. Result (Accepted/rejected, ran too long)
4. depth of tree
5. Number of configurations explored
6. average nondeterminism</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries:

Language: Python
Libraries: csv, os (Used for parsing and file creation)</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):

Hashmaps (dictionaries): Used to store the NTM definition (states, transitions, tape, etc)
Arrays (nested lists): Used to store the config while executing BFS, in a tree structure. The k+1 element of the outer list is all the possible configurations reached from those in the kth element of the list.</td></tr>
<tr><td>10</td><td>General operation of code (for each subproject):

The main function starts off by reading the test cases from the “data_test_cases_Sajith.txt”, with which each line in the test case consists of a a string that is parsed into “input string (tape)”, “depth_limit”, and the “file_name” for the specific machine being simulated from the inputs folder.

Then the simulate_ntm function opens the specified CSV file and parses it into a dictionary representation of the Turing Machine (TM). The dictionary uses keys for header-line information (e.g., start state, accept state) and stores each state with its possible transitions as a list of values.

Then the ntm_bfs function is then called with the TM dictionary and input tape. It builds</td></tr>
</table>

	a configuration tree, represented as a nested list, where each level corresponds to configurations derived from the starting state and input. The tree begins at level 0 with the starting configuration. This BFS loops through the current level, generating all possible configurations for the next level based on the transitions defined in the TM. Everytime a transition hits the accept state the loop terminates. Each level is tracked using the variables in the loop. The code handles transitions by iterating over the transitions for the current state: If the input matches the transition condition, a new configuration is created for the next level by moving the tape head and writing the updated tape. If no valid transitions exist, a reject configuration is added to the next level. The code checks if non-reject configurations remain. If none are left, the BFS ends. If the specified depth limit is reached, the loop also terminates. Once the BFS is complete, the function returns the configuration tree and the reason for termination (accept, reject, or timed_out). It then calculates metrics such as the average nondeterminism by dividing the total transitions by the number of configurations in the outermost list. Finally, the configuration tree for each case is outputted into a file number by machine_{number}_test_{test case number for that machine} into the output directory.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. I used 14 test cases derived from six Turing Machine definitions. For each Turing Machine, I included one test case for acceptance and one for rejection, as determined by the language definition. These test cases were specifically designed to be nontrivial, ensuring that the program correctly identifies whether a string should be accepted or rejected. While I experimented with various additional strings to confirm correctness, I provided a total of 13 strings in the input file for simplicity. Additionally, I included a single very long string to test the program's ability to enforce the execution depth limit (machine_3_test-4). The Turing Machines used in my tests include three nondeterministic and three deterministic ones, with each nondeterministic machine paired to a deterministic equivalent. This setup enables testing for both nondeterministic and deterministic behavior. It ensures the program delivers consistent accept/reject results and that the measured nondeterminism varies as expected between the tests (with deterministic machines having a nondeterminism value of 1.0). The TM definitions were used as inputs in the data folder (6 as mentioned earlier) and the data_test_cases_Sajith.txt contained the 14 specific cases tested. Each of the outputs of the test cases are named as mentioned earlier machine_{number}_test_{test case number for that machine}

12	How you managed the code development: I managed the code development by understanding the primary goals of each of the sub functions that I needed to make: a test parser, a table maker, an outputer and a solver. Additionally, I used VS Code to have a good control over my directory of input and output files so I could quickly make any changes that were required. Once I had it setup, I began working on the main functions for the programs and then attempted to tie them all together inside the main function and pipeline them to output directories.
13	Detailed discussion of results: The results demonstrate that measured nondeterminism increases in Turing Machines (TMs) with a higher number of nondeterministic transitions (multiple transitions for the same input in a single state). This occurs because the program adds more options from the transition list for a given state and input to the next level of the configuration list, thereby increasing the average nondeterminism. For example, the nondeterminism in the *a+* (data_1_machine_Sajith.csv) TM is 2.2 (and 2.0 in a second test), while the *a*b*c** (data_3_machine_Sajith.csv) TM has a higher nondeterminism of 5.55 (and 5.0 in a second test). Across multiple tests, including those with the same input string length, *a*b*c** consistently shows higher nondeterminism because it has more transitions available for the same input and state. Specifically, *a*b*c** contains 13 nondeterministic transitions, whereas *a+* only has 2. Furthermore, longer input strings result in higher nondeterminism. For instance, a 53-character (output_machine_3_test-4_Sajith.txt) input to *a*b*c** raises nondeterminism above 8.0. Finally, as a control, fully deterministic machines (i.e., those with entirely unique transitions) consistently yield a measured nondeterminism of 1.0.
14	How team was organized: Individual (solo team)
15	What you might do differently if you did the project again: I would try to implement more helper functions instead of having 2 large functions as that would make debugging much easier. I'm glad I did so with my table maker and output file generation.
16	Any additional material: Table Image:

	Machine Name	Input String	Result	Depth	Total # Configurations Simulated	Average Non-Determinism
2	a+ Nondeterministic	aaa	accepted	5	11	2.2
3	a+ Nondeterministic	aab	rejected	4	8	2.0
4	a+ Deterministic	aaa	accepted	4	5	1.0
5	a+ Deterministic	aaab	rejected	5	5	1.0
6	a*b*c* Nondeterministic	aabcccc	accepted	9	50	5.555555555555555
7	a*b*c* Nondeterministic	abc	accepted	5	25	5.0
8	a*b*c* Nondeterministic	aabdecccc	rejected	5	26	5.2
9	a*b*c* Nondeterministic	aaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbcccccccccccc	timed_out	51	427	8.211538461538462
10	a*b*c* Deterministic	abccccd	rejected	8	8	1.0
11	a*b*c* Deterministic	abcccc	accepted	7	8	1.0
12	{t | t has the same number of 0's and 1's} Nondeterministic	01011100	accepted	44	68	1.511111111111111
13	{t | t has the same number of 0's and 1's} Nondeterministic	01011101110	rejected	53	75	1.4150943396226414
14	{t | t has the same number of 0's and 1's} Deterministic	01011100	accepted	43	44	1.0
15	{t | t has the same number of 0's and 1's} Deterministic	010111001011	rejected	66	66	1.0