Project 2 Readme Team Kyle Team

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: Kyle Team</td></tr>
<tr><td>2</td><td>Team members names and netids: Kyle Crosby, acrosby2</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: Successful</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 11</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/kylecrosby2/Project2Theory</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_kyleteam.py</td><td>Python code to trace a Turing machine. Input from stdin is formatted as "string depth_limit test_file_name"</td></tr>
<tr><td colspan="2">Test Files</td></tr>
<tr><td>check_input_kyleteam.txt
data_a_plus_DTM_kyleteam.csv
data_a_plus_kyleteam.csv
data_abc_star_DTM_kyleteam.csv
data_abc_star_kyleteam.csv
data_equal_01s_DTM_kyleteam.csv
data_equal_01s_kyleteam.csv
data_sample_ntm_1_kyleteam.csv
sample_ntm_2_a_plus.csv</td><td>check_input is a text file to send to stdin when running traceTM_kyleteam.py. Each line runs a test machine with varying cases. Names files from the "tests" directory.

data files are in the "tests" directory. They are CSVs defining the Turing Machines.</td></tr>
<tr><td colspan="2">Output Files</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td>output_kyleteam.txt</td><td>Output of traceTM_kyleteam.py piped to a text file. Contains information on each tree tested and the amount of nondeterminism.</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td></td><td></td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries: Python, csv library, sys library</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):
Dictionary to store the Turing Machine after reading from the CSV file. Dictionary keys are the header lines with values containing their information, and are also state names with values as lists of lists of the possible transitions from that state.
Nested list (tree) of configurations for the given TM and tape. The k+1st element of the outer list is a list of 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 reads stdin line by line, with each line formatted as “string depth_limit test_file_name” and splits the line and sends the data as arguments to the simulate_ntm function.
simulate_ntm opens the given CSV file and reads each line to create a dictionary representing the TM with keys for the header line information and keys for each state corresponding to a value as a list of possible transitions from the state.
Next, ntm_bfs is called with the TM dictionary passed to it. ntm_bfs creates a tree (nested list) of configurations from the given TM and input tape. The tree starts at level 0 with the starting configuration derived from the starting state in the dictionary. The tree is grown through a while loop that loops through each level and adds all possible configurations from the current level to the next level. The while loop stops if any of the transitions it reads from the dictionary contain the accept state. There are two variables tracking the current level and index within the level, so the while loop knows when it is done looping through one level and ready to append a new one to the tree. The possible configurations are accessed by looping through each transition in the list corresponding to the next state in the current transition (if the input matches). There is a short if else tree to properly move the head and write new input for the new-level configurations. A reject state configuration is added to the next level if no transition is ever made with the current input. After each level is looped through, it is checked to see if there are any non-reject states. If there are none, then the BFS ends, otherwise it continues until reaching the specified max level limit. Once the loop is done, the function returns the configuration tree and how the loop ended (accept, reject, or limit reached).
Next, all of the information is output based on the configuration tree. Nondeterminism is measured by summing the number of transitions and dividing by the length of the outer list to get average nondeterminism. Last, the configuration tree is printed.</td></tr>
</table>

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 from six Turing Machine definitions. For each TM, I provide one test case for acceptance and one for rejection based on the language definition. Each accept/reject string is nontrivial and shows if the program is outputting correctly based on whether it accepts/rejects (I also tried various other strings to verify, but provided 13 in my test input file for simplicity). I additionally provide one very long string to verify that the execution limit functionality works. The Turing Machines for my tests are three nondeterministic and three deterministic (each corresponding to one another). This allows testing of strings of both nondeterministic and deterministic to verify that the program gives the same acceptance/rejection output, and that the measured nondeterminism varies between the tests (with the deterministic machine being 1.0).
12	**How you managed the code development**: Developed code in VSCode locally. Named output files iteratively to track changes. Saved deleted functions to a temporary file in case of later use. Created a GitHub repository when finished and pushed all files.
13	**Detailed discussion of results**: As seen in the code output, measured nondeterminism increases with test TMs that have more non-deterministic transitions (multiple transitions for the same input in a single state). In the program, this corresponds to more options in the possible transitions list for a given state and input, which then is added to the next level of the configuration list, thus increasing the average nondeterminism. This is demonstrated by comparing the nondeterminism in the a_plus NTM and the abc_star NTM. The measured nondeterminism in a_plus is 2.0 (and 2.2 for a second test), while abc_star is 5.5 (and 5.2 for a second test). Upon multiple tests (including for the same input string length), abc_star always has higher nondeterminism. This is because there are more transitions in the machine that have the same input and same state. There are 13 non-deterministic transitions in abc_star, while a_plus only has 2. Additionally, we can observe that longer input strings create more non-determinism. The 53-character long input to abc_star increases non-determinism above 8.0. Finally, as a control we observe that fully deterministic machines (i.e. have all unique transitions) have measured nondeterminism of 1.0.
14	**How team was organized**: Solo
15	**What you might do differently if you did the project again**: I would create a GitHub prior to starting the project to provide version control.
16	**Any additional material**: No