Project 2 Readme Team KB

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: KB</td></tr>
<tr><td>2</td><td>Team members names and netid: Kaila Bryant (kbryant3)</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: Tracing NTM</td></tr>
<tr><td>4</td><td>Overall success of the project: The project was a success! My code worked on different test cases; accepting when appropriate and rejecting when necessary.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 9-10 hours</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/kailabryant/Project2-TOC</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>ntm_tracer.py</td><td>This file contains the main code for the NTM tracer. The code in this file was the driver code that contained the BFS algorithm of the NTM config tree.</td></tr>
<tr><td>turing_machine.py</td><td>This file contains the helper code that in used to make ntm_tracer.py</td></tr>
<tr><td colspan="2">Test Files</td></tr>
<tr><td>aplus.csv</td><td>This csv file contains the information for a NTM that recognizes all strings of any # of a's and rejects the empty string. This file was used to test the functionality of ntm_tracer.py.</td></tr>
<tr><td>composite.csv</td><td>This csv file contains the information for a NTM that recognizes strings of 1's with the length of the string being a composite number. This file was used to test the functionality of</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td></td><td>ntm_tracer.py.</td></tr>
<tr><td>palindrome.csv</td><td>This csv file contains the information for a NTM that recognizes all strings that are palindromes over the alphabet a,b. This file was used to test the functionality of ntm_tracer.py.</td></tr>
<tr><td colspan="2">Output Files</td></tr>
<tr><td>aplus_output_kb.png</td><td>Shows the output of running ntm_tracer.py on aplus.csv on the input “aaa” The output shows the tree depth, total transitions, and how many steps it took for the machine to accept the string.</td></tr>
<tr><td>composite_output_kb.png</td><td>Shows the output of running ntm_tracer.py on composite.csv on the input “111111” The output shows the tree depth, total transitions, and how many steps it took for the machine to accept the string.</td></tr>
<tr><td>palindrome_output_kb.png</td><td>Shows the output of running ntm_tracer.py on palindrome.csv on the input “aabaa” The output shows the tree depth, total transitions, and how many steps it took for the machine to accept the string.</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td>N/A</td><td>N/A</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries:
I used python for my project and indirectly used the libraries:
- csv
- sys
- These libraries were used in the helper code src/helpers/turing_machine.py and I imported turing_machine in order to write my code in ntm_tracer.py</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project): (only worked on ntm_tracer.py)
The key data structures were:
- Lists
- I used a list to represent a single state of the Turing machine, <code>[left, state, right, parent, transition_index]</code>
- List of lists
- I used a list of lists in order to represent the computation tree in order to track the entirety of the nondeterministic computation which helped when I needed to backtrack to construct accepting paths
- Dicts</td></tr>
</table>

	- I used dictionaries to represent transitions, using dictionaries with keys like `'next', 'write',` and `'move'` helped in defining the NTM;s nondeterministic behavior - Tuples - I used single element tuples for `read_symbols → (current_char,)` to prepare the tape symbols in transition lookup
10	**General operation of code (for each subproject)** run(self, input_string, max_depth) - This function was the main execution that performed the BFS traversal of the NTM. It worked by first creating an initial config with an empty left tape, start state, and input string. Then it looped, processing each configuration by level until the max_depth was reached. At each configuration it checked for acceptance or rejection and generated children by applying valid transitions to create the next-level configurations. For each valid transition, it wrote a symbol to tape the head position, then moved the head left or right (updating the left and right strings accordingly), and then created a new configuration with a parent to backtrack. This function also handled termination; if accepted it would print the accepting path and tree, if rejected it would print the rejection with depth, and if the max_depth was reach, it would print to the user the incomplete exploration. print_trace_path(self, tree, final_config) - The purpose of this function was to backtrack from an accepting configuration to the root in order to reconstruct the successful computation path. It did so by starting at the final_config and followed the parent pointers (depth, index) back all the way to the root. Then it would build a list of configs from the accept state back to the start. Finally, it would print each config in order from start to accept Print_config(self, config) - This function acted as a helper function; it essentially aided in formatting and printing a single config in a readable manner. It would extract left, right, and state from the config. Print_tree(self, tree) - This function printed the entire computation tree, level-by-level. It would iterate through each level of the tree and print all configurations at that level with their index, using the print_config function.
11	**What test cases you used/added, why you used them, what did they tell you about the correctness of your code.** I used the string “aaa” on the input file aplus.csv, I used this to verify that my ntm_tracer.py code worked because the machine should accept all strings of any # of a’s and reject the empty string and the results of this test case demonstrated that the search algorithm found the accepting path and showed how it got that path. Next, I used the string “111111” on the input file composite.csv; with this test case, the machine should accept strings of 1’s with a length of a composite #. The results of this test again showed that my code was correct because the output of my code matched the logic of the NTM and correctly traced the accepting path. Lastly, I used the string “aabaa” on the input file palindrome.csv. I used this code to demonstrate a familiar behavior of an NTM.
12	**How you managed the code development** Given that I worked alone, I solely managed the development of the code. I split up the code throughout multiple days. I first worked on developing pseudocode for the overall

	functionality of the program and hand-traced examples to ensure the logic translated from paper to code. After I felt confident in my logic, I tried coding the main functionality before focusing on formatting the output, including backtracking to print the entire computation tree
13	**Detailed discussion of results**: All my results demonstrated the functionality of my code. For example, in the case of the palindrome example, we see that the tree depth was 2 and the total transitions was 5; this tells us that relatively few branches were explored and only a shallow exploration was necessary. Another example was the test case on the aplus machine. On the input 'aaa', the tree depth was 4 and the total transitions were 7. The machine processed left to right as it consumed each 'a' and we can see from the results that there was a parallel exploration of multiple paths, at level 1 we have 2 branches (q1 and q2) paths.
14	**How team was organized** Since I worked alone, I was responsible for writing all the code in ntm_tracer.py and figuring out what csv files I was planning on using in order to test the functionality of my code.
15	**What you might do differently if you did the project again** If I did the program again, I would've spent more time looking at the helper code first. I originally struggled to write my code in ntm_tracer.py and wrote useless code that the helper code could've helped with. Once I actually took time to read through turing_machine.py, it was far easier to write my code and leverage the helper functions that were written for us.
16	**Any additional material:** Overall, the project was an interesting way to apply what we have covered in class in a code format and it was useful to see the full computation tree printed and the path a given NTM took given a certain input.