Project Readme Template

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: Trigonometry</td></tr>
<tr><td>2</td><td>Team members names and netids: Ella Trigiani etrigian</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: Tracing NTM Behavior</td></tr>
<tr><td>4</td><td>Overall success of the project: I think very successful, since I have run my code on lots of different inputs and they appear to be working.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: ~10 hours</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/ellatrigiani/Project2-TOC.git</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>src/ntm_tracer.py</td><td>• Main NTM_Tracer class implementation from TuringMachineSimulator
• Step() method handles single-tape transition logic for R,L, and S movements
• Implements tape boundary handling (leftmost position returns BLANK when moving left)
• run() method performs BFS of NTM configuration tree
  ○ Tracks parent relationships using dictionary for path reconstruction
  ○ Handles accept and reject states</td></tr>
</table>
</td></tr>
</table>

	and implicit rejection (missing a transition) ● print_trace_path() backtracks through parent pointers to display accepting computation path ○ Configuration format is [left, state, right] where right includes head symbol
src/helpers/turing_machine.py	● Given to us to use ● Provides common TM infrastructure ● load_machine() parses CSV files to extract states, alphabet, and transitions ○ Read in all states, names, sigma, gamma, etc. ● get_transitions() returns all valid transitions for a given state (state, symbol) pair
Test Files	
input/palindrome.csv	● NTM recognizing one or more a's ● Tests basic non determinism since two transitions on same symbol a
input/0n1n.csv	● NTM for recognizing 0^n1^n ● Tests complex marking algorithm: masks pairs of 0s and 1s with X, returns to start and repeats ● Tests proper leftmost boundary handling
input/composite.csv	● Tests if length of string of 1s is a composite number ● Demonstrates nondeterminism in choosing certificate,

<table>
<tr><td></td><td>
<table>
<tr><td></td><td>followed by deterministic verification</td></tr>
<tr><td>input/aplus.csv</td><td>● NTM recognizing palindromes over the alphabet</td></tr>
<tr><td colspan="2">Output Files</td></tr>
<tr><td>output/palindrome_output_trigonometry</td><td>● Shows output from palindrome tests on one successful string and one reject string
● Shows the trace results</td></tr>
<tr><td>output/0n1n_output_trigonometry</td><td>● Shows the output from 0n1n on one successful string and one reject string
● Shows the trace results</td></tr>
<tr><td>output/composite_output_trigonometry</td><td>● Shows success when entering string “111111” which was given to us and shows tracing result</td></tr>
<tr><td>output/aplus_output_trigonometry</td><td>● Shows success when entering “aaa” which was given to us and shows tracing result</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:

Language: I used python for the entire project

Libraries:
1. csv - this was in the helper functions to parse the input CSV files
2. sys - this was also used in helper functions to be able to read command line arguments
Note: I didn’t have to import any additional libraries than the ones given to us</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):

Program 1: NTM Tracer (the only project I did, since one person group)

1. Configuration representation: List written as [left, state, right]</td></tr>
</table>

	a. Left (string): all tape symbols to the left of the head b. State (string): current state name c. Right (string): symbol under head + all symbols to the right 2. Tree Structure: List of lists written as [[config1],[configa, configb],...] a. Outlet list: each element represented one depth level b. Inner lists: all possible configurations at that depth c. Enables BFS exploration later on 3. Parent Tracking: Dictionary {tuple(child_config): tuple(parent_config)} a. Keys: child configurations as tuples since they need to be hashable b. Values: parent configurations c. Enables path reconstruction from accept state back to start state 4. Transition dictionary: transitions[state] = [{read, next, write, move},..] a. Allows multiple transitions per (state, symbol) pair for nondeterminism
10	General operation of code (for each subproject) Program 1: NTM Tracer (the only project I did, since one person group) 1. Initialization: a. Load machine definition from CSV file using helper functions given b. Create initial configuration [“”, start_state, input_string] c. Initialize tree with single level containing initial config d. Set depth = 0 2. BFS Loop (while depth < max_depth and not accepted and tree[depth]): a. Get all configurations at current depth level b. For each configuration: i. Check termination: if in an accept state, then print the path and stop ii. Check rejection: if in a reject state, skip this branch iii. Get transitions: use the get_transitions function to find all valid moves iv. Handle the case where there are no transitions, which means reject and continue v. Generate children for each valid transition: 1. Apply transition using step() method 2. Create new configuration 3. Add to next level 4. Track parent relationship c. If all the branches reject, then print rejection and stop d. Otherwise append next level to tree and increment depth 3. Transition application (step() method): a. Right move: add written symbol to the left and move the head to the next position b. Left move: take rightmost character from the left as new head position (or BLANK if at leftmost), preserve what was there c. Stay move: just replace symbol under head d. Return updates left and right strings 4. Path Reconstruction (print_trace_path()): a. Start from accepting configuration b. Follow parent pointers backwards to initial configuration

	c. Reverse the path to get start to end order d. Print each configuration
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. 1. A^+ (aplus.csv) a. Should accept all strings consisting of any number of a's, and should reject the empty string b. The purpose was to verify that my NTM code worked, especially the nondeterministic aspect in choosing the last a c. The results confirmed that the BFS explored multiple paths and found an accepting path, which indicates the code was running correctly. 2. 0^n1^n (0n1n.csv) a. Should accept strings of n number of 0s followed by n number of 1s, and reject everything else b. The purpose of this is the same as above but slightly more complex. This check I actually struggled with since the given machine wasn't functioning properly. c. This showed further that my code could run on other cases. 3. Palindrome (palindrome.csv) a. Should accept strings that are palindromes and reject ones that are not b. Similarly the purpose was to check if the NTM code was functioning properly which it was, and this reassured that. 4. Composite number testing (composite.csv) a. Should accept strings of 1s that the length is a composite number, and reject everything else. b. Again this was another test case I decided to try since it had slightly different functionality than the previous ones. This being correct again solidified the validity of my code. I also made sure within each kind of test to try examples that should work and shouldn't work and cross validate them. It was also important to include edge cases like the empty string and understand in which languages should that pass versus fail. The key thing I learned was in the step() method that handling the leftmost boundary was very important, otherwise the code would not run properly. Also, parent tracking with tuples, not lists, since they are hashable was needed for path reconstruction, and was something I got errors on at first.
12	How you managed the code development Since I was working alone I was able to build out the entire program and really understand every step. I started with the skeleton code provided and implemented the BFS search steps. I then created step() which is a helper function that isolates the tape manipulation logic, which I found to be the most difficult part. I then filled in the print_trace_path() function to properly output the other moving pieces of the program. One of the most important aspects of the project, in my opinion, was testing. So I first used very simple machine inputs to test if my code was working properly, then advanced to more complicated, then tested edge cases. I ran into a few bugs along the way and took time to figure them out.

	It was important to me to break each of the steps into different pieces so that the entire code was more digestible. I attempted to break down each function into one category of action which could be easily read. Overall it was a very fun project to code.
13	Detailed discussion of results: The NTM tracer successfully implements a breadth-first search exploration of nondeterministic Turing machine computation trees. The program tracks depth and number of transitions. For example, testing the a^+ machine on input “aaa” resulted in a depth of 4 with 15 total transitions. The main challenge was handling the leftmost boundary. Initially moving left and left=”” made infinite loops, which didn’t work. The solution I came up with was to set moved_to = BLANK, breaking the infinite loop. This was verified through test cases that were previously looping, then ending and either accepting or rejecting. The validation of the program was tested manually by me tracing through accepting paths printed by the program. The accepting path for “aaa” on a^+ showed the correct sequence of configurations from start to accept state. And by using the parent tracking method, reconstructing and recording the paths was fairly straightforward. The BFS approach was needed, since it guarantees finding an accepting path if they exist, unlike DFS which could loop forever on one branch.
14	How the team was organized: Since I was working alone, it was just me doing all the parts.
15	What you might do differently if you did the project again: Honestly this project went much smoother than the last one. I think if I were to add two things I would better consider optimization and add more test cases. Firstly, better optimization is good coding practice and would’ve been interesting to think about. Secondly, I believe I thoroughly tested my code but there is always room for additional cases and I think it would’ve been useful to check even more using our old homeworks and other problems. But since mine already worked for various problems, it wasn’t necessary.
16	Any additional material: No. Overall I really liked this project!