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: Czapewski</td></tr>
<tr><td>2</td><td>Team members names and netids: Jason Czaplewski - jczaplew</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 project; I was able to create, test, and output the necessary requirements for this project.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 6-7 hours</td></tr>
<tr><td>6</td><td>Link to github repository: 11jac11/-TracingNTMBehavior_Czaplewski: This is the repository for Jason Czaplewski's second theory of computing project</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><th>File/folder Name</th><th>File Contents and Use</th></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td>traceTM_jczaplew.py</td><td>This Python program implements a Turing Machine (TM) simulator that reads machine definitions from CSV files and processes input strings according to the specified rules. The simulator traces the execution steps of the TM, showing how the machine reads and modifies its tape while moving through different states. It handles both deterministic and non-deterministic transitions, keeping track of all possible execution paths until it finds an accepting path or determines the input should be rejected.
The program takes two command-line arguments: a CSV file containing the TM's definition and an input string to process. The CSV file must specify the machine's name, states, alphabets, and transition rules. During execution, the program maintains a configuration tree to track the machine's state, tape contents, and head position at each step. It also counts the total number of transitions made during the computation.
As the TM runs, it provides detailed output showing each step of the computation, including the current state, tape contents, and available transitions. The final result, whether the string is accepted or rejected, is displayed both in the console and</td></tr>
</table>
</td></tr>
</table>

	written to an output file, along with the complete execution path if the string is accepted.
Test Files	
aplus.csv	The aplus.csv file defines a Turing Machine that accepts any string containing at least one 'a', regardless of other characters. The machine reads through the input string and accepts as soon as it finds an 'a', continuing to the end of the string. For example, it accepts strings like "a", "bba", "aaaa", or "bbbaaa", but rejects strings like "bbb" or empty strings.
ends_in_01.csv	The ends_in_01.csv file defines a Turing Machine that accepts binary strings ending with "01". The machine reads through the input string from left to right, and only accepts if it finds "01" at the very end of the string. All other strings, including those containing "01" in other positions but not at the end, are rejected. For example, it accepts strings like "01", "101", "1001" but rejects strings like "10", "010", or "0110".
equal_01.csv	The equal_01.csv file defines a Turing Machine that accepts binary strings containing an equal number of 0s and 1s. The machine keeps track of the balance between 0s and 1s as it reads through the input string, accepting only when it reaches the end and has seen the same count of each digit. For example, it accepts strings like "01", "1100", "101010", but rejects strings like "100" or "111".
Output Files	
aplus testing a.png aplus testing aaa.png aplus testing bbb.png	These files show the command line input of “a”, “aaa”, and “bbb” being inputted into the terminal. It then shows the output that each of these inputs gives.
ends_in_01 testing 01.png ends_in_01 testing010.png	These files show the command line input of “01” and “010” being inputted into the terminal. It then shows the output that each of these inputs gives.
equal_01 testing 010.png equal_01 testing 10.png equal_01 testing 1010.png	These files show the command line input of “010”, “10”, and “1010” being inputted into the terminal. It then shows the output that each of these inputs gives.
Plots (as needed)	
NOTE: Here is where I included 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
7. comments

NTM aplus:

- When I used the string "a" I got an ACCEPT result with 1 depth of tree and 2 transitions explored. The average nondeterminism is 2 (2 transitions / 1 steps). The machine quickly found an 'a' and accepted.
- When I used the string "aaa" I got an ACCEPT result with 3 depth of tree and 4 transitions explored. The average nondeterminism is 1.33 (4 transitions / 3 steps). The machine found an 'a' immediately but had to read to the end.
- When I used the string "bbb" I got a REJECT result with 3 depth of tree and 3 transitions explored. The average nondeterminism is 1 (4 transitions / 4 steps). The machine had to read all 'b's before rejecting since no 'a' was found.

Comments: The machine is mostly deterministic, with clear paths for 'a' and 'b' inputs. It accepts as soon as it finds an 'a', but must read to the end if seeing 'b's. The number of transitions grows linearly with input length

NTM ends_in_01:

- When I used string "01" I got an ACCEPT result with 3 depth of tree and 6 transitions explored. The average nondeterminism is 2 (6 transitions / 3 steps). The machine found the "01" pattern at the end and accepted.
- When I used string "010" I got a REJECT result with 3 depth of tree and 9 transitions explored. The average nondeterminism is 3 (9 transitions / 3 steps). The machine rejected because even though it found "01", the string continued with "0" at the end.

Comments: The machine shows some nondeterminism when reading "0" as it must decide whether to start checking for the "01" pattern. The machine must read the entire input to ensure "01" is at the end. Longer strings require more transitions as the machine must check every position

NTM equal_01:

- When I used string "010" I got a REJECT result with 3 depth of tree and 3 transitions explored. The average nondeterminism is 1 (3 transitions / 3 steps). The machine rejected because it found two 0s but only one 1.
- When I used string "10" I got a REJECT result with 2 depth of tree and 3 transitions explored. The average nondeterminism is 1.5 (3 transitions / 2 steps). The machine rejected because it found one 1 and one 0, but no balancing digit.
- When I used string "1010" I got an ACCEPT result with 4 depth of tree and 5 transitions explored. The average nondeterminism is 1.25 (5 transitions / 4 steps). The machine accepted because it found two 1s and two 0s.

Comments: The machine maintains a count by switching between states. Each transition is mostly deterministic once in a counting state. The machine must read the entire input to ensure equal counts. Longer balanced strings require more transitions but maintain similar nondeterminism ratios

8	Programming languages used, and associated libraries: I created the project using Python as the programming language on the student machines. It uses Python's standard libraries including csv for reading machine definitions, sys for command-line argument processing, and typing for type hints. The collections module provides the defaultdict data structure for managing transitions, while os handles file operations. No external libraries were required beyond these built-in modules.
9	Key data structures (for each sub-project): The project primarily relies on a TMConfig class that stores machine configurations using strings for tape contents, states, and head positions, while tracking parent configurations for path reconstruction. The core TuringMachine class uses a defaultdict to manage transition rules, maintaining lists of configurations organized by tree levels. Supporting data structures include lists for execution paths, dictionaries for transitions, strings for tape contents and state names, and sets for alphabets and valid states. These structures work together across all three machines (aplus, ends_in_01, equal_01), with each machine implementing different transition rules and states while sharing the same underlying architecture.
10	General operation of code (for each subproject): The code operates by first reading a Turing Machine definition from a CSV file, which specifies states, alphabets, and transition rules. When processing input strings, it creates an initial configuration and goes through possible transitions. The machine tracks its current state, tape contents, and head position while following transition rules during execution. When an accepting state is reached or all paths are rejected, the program outputs the result along with statistics about the computation, including the number of transitions and depth of the search tree. This operation remains consistent across all three machines, though each implements different acceptance criteria through their specific transition rules.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code: I tested the three Turing Machines with various carefully selected inputs to verify their correctness: - For aplus.csv, I tested with "a", "aaa", "bbb", and empty string. These cases verified that the machine correctly accepts strings containing at least one 'a' regardless of position, while properly rejecting strings with no 'a's. The varying lengths helped confirm proper tape traversal. - For ends_in_01.csv, I tested with "01", "010", "1010", and "10". These cases confirmed that the machine only accepts when "01" appears at the end, properly rejects when "01" appears elsewhere or is incomplete, and correctly handles different string lengths. - For equal_01.csv, I tested with "1010", "010", "10", and empty string. These inputs verified that the machine correctly counts and balances 0s and 1s, accepts only when counts are equal, and properly handles edge cases like empty strings and unbalanced inputs. Each test case revealed different aspects of the machines' behavior, helping verify transition handling, state management, and proper termination conditions. The varying input lengths and patterns also helped confirm that the tape navigation and non-deterministic path exploration worked correctly.

12	How you managed the code development: I developed the code incrementally, starting with the core TM simulator structure and basic functionality. Each machine's transition rules were developed and tested separately, beginning with simple cases before doing the rest. Regular testing helped identify and fix bugs while maintaining code quality. The process focused on clean implementation, proper documentation, and consistent error handling throughout development.
13	Detailed discussion of results: The Turing Machine simulator successfully implemented three different machines with distinct behaviors. The aplus machine efficiently recognized strings containing 'a', showing consistent performance with both accepting and rejecting cases. The ends_in_01 machine accurately identified strings ending in "01", demonstrating proper pattern matching at string endings. The equal_01 machine correctly tracked balanced counts of 0s and 1s, handling various input lengths effectively. The output showed reasonable transition counts and tree depths across all machines. Non-deterministic branching was handled efficiently. The output format provided clear visibility into machine operations. All three machines demonstrated correct behavior across various test cases, including edge cases and invalid inputs, confirming the robustness of the implementation.
14	How the team was organized: Since I worked by myself on this project, I independently worked on all the parts including design, implementation, testing, and documentation. While working alone provided advantages like quick decision-making and consistent coding style, the project could have benefited from team collaboration through code reviews, parallel development, and more comprehensive testing. Working in a group would have brought different perspectives and allowed for shared documentation responsibilities. Additionally, I procrastinated the work a decent amount so working in a group could have prevented this and allowed for me to get the code done quicker.
15	What you might do differently if you did the project again: If redoing this project, I would make the code structure simpler and add better testing from the start. I would improve how the program shows its results and add clearer error messages. Better comments and documentation would make the code easier to understand. This would allow for the program to be more user-friendly and easier to maintain.
16	Any additional material: I encountered some difficulties implementing proper non-deterministic transitions and managing multiple configurations. Debugging the state transitions proved particularly challenging, especially for the equal_01 machine where maintaining accurate counts required careful state management. Another significant difficulty was ensuring the correct handling of tape boundaries and special cases like empty strings. The output formatting also presented challenges in clearly displaying the machine's execution steps while keeping the information concise and readable. Despite these difficulties I was able to successfully get the code to work.