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"

1	Team Name: RED
2	Team members names and netids: Rogelio Diaz rdiaz3
3	Overall project attempted, with sub-projects: Program 1: Tracing NTM Behavior. I implemented a Nondeterministic Turing Machine (NTM) simulator that traces all possible execution paths using a Breadth-First Search (BFS) approach. Key features include, Parsing standard CSV machine definitions as specified in the handout. Simulating simultaneous branches of execution without backtracking using a "List of Lists" state tracking method.Calculating the "Degree of Nondeterminism" and handling implicit rejection. Backtracing and printing the specific "winning path" (transition log) for accepted strings.
4	Overall success of the project: The NTM Tracer functions correctly across all tested edge cases. Nondeterminism: It successfully branches execution flow for inputs with multiple valid paths (verified with aplus.csv). Tape Movement: It correctly handles bidirectional tape movement and boundary conditions (verified with palindrome.csv). Termination: It correctly identifies the first accepting configuration to minimize execution time, or correctly detects when all branches effectively die (rejection). Metrics: The output accurately reports the depth, total transitions, and the calculated degree of nondeterminism.
5	Approximately total time (in hours) to complete: 8 hours

6	Link to github repository: https://github.com/red849/Project2-TOC
7	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.

File/folder Name	File Contents and Use
Code Files	
src/ntm_tracer.py src/helpers/turing_machine.py	src/ntm_tracer.py: Main implementation of the BFS trace logic, transition engine, and statistics reporting. src/helpers/turing_machine.py: Helper class responsible for parsing the CSV header/body and handling wildcard matching (*).
Test Files	
/input/aplus.csv /input/composite.csv /input/palindrome.csv	/input/aplus.csv: Checks basic nondeterministic branching (Right moves) for the language a+. /input/composite.csv: Checks complex movement (Left/Right/Stay) and correct handling of the left tape boundary. /input/palindrome.csv: Checks "guess and verify" logic (implicit rejection) for detecting composite numbers.
Output Files	
Terminal	The program prints the level-by-level configuration trace, the specific "winning path" backtrace, and the final statistical metrics directly to the console.
Plots (as needed)	
N/A	

8	Programming languages used, and associated libraries: Python csv Src.helpers.turing_machine
9	Key data structures (for each sub-project): List of Lists (The Tree): The primary structure for BFS. tree[k] stores a list of all active configurations at depth k, allowing us to simulate parallel execution. Configuration List: Each machine state is tracked as [left_tape, current_state, right_tape, parent_index, transition_info]. The parent_index was added to allow backtracing the winning path from leaf to root. Transition Dictionary: A dictionary mapping (state, read_char) to a list of possible outcomes. This provides O(1) lookup speed even when multiple transitions exist for the same input.
10	General operation of code (for each subproject): Initialization: The CSV is parsed into the dictionary, and the tree is initialized with the start state at Level 0. Breadth-First Loop: The code iterates through every configuration in the current level: Check Accept: If qacc is found, the loop breaks immediately. The "winning path" is reconstructed by following parent pointers backwards to the start. Check Reject: If qreject is hit or no valid transitions exist (implicit reject), that branch is abandoned. Expand: If the branch is alive, the code looks up valid moves (including wildcards) and appends the resulting configurations to the next level's list. Termination: Execution stops when an accept state is hit, the queue becomes empty (all paths rejected), or the max depth is exceeded.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code.: N/A

12	How you managed the code development: I used the uv package manager for environment handling. Development was incremental: 1. Verified the CSV parser correctly read the 7-line header and transitions. 2. Implemented the basic BFS loop to print configurations level-by-level. 3. Added the metrics logic (Transitions / Non-Leaves). 4. Implemented the "Parent Pointer" logic to enable the execution trace back (Section 4.2). 5. Verified output formatting against the project PDF requirements.
13	Detailed discussion of results: The tracer successfully computed the required metrics. For deterministic inputs, the "Degree of Nondeterminism" stayed at 1.0. For the aplus machine, the degree was ~1.88, which matches the expected behavior of a machine that splits execution at nearly every step. The "winning path" output effectively filters out the noise of failed branches, showing only the correct sequence of transitions that led to acceptance.
14	How the team was organized: Single-person team. I was responsible for all implementation, testing, and documentation.
15	What you might do differently if you did the project again: I would implement a verbose flag to suppress printing every configuration for deep trees (depth > 50), as the terminal output becomes difficult to read for complex problems like the composite number check. I might also consider adding a Graphviz visualization to plot the execution tree.
16	Any additional material: