Project x Readme Team cander35

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: cander35
2	Team members names and netids: Cate Anderson (cander35)
3	Overall project attempted, with sub-projects: Program 1: Tracing NTM Behavior
4	Overall success of the project: I think the project was quite successful as I think I was able to accomplish the tasks and goals of Program 1: Tracing NTM Behavior. Parsing the data and organizing the output definitely challenged me, but once I got over that hurdle things went quite smooth.
5	Approximately total time (in hours) to complete: 7 hours
6	Link to github repository: https://github.com/cateranderson/TOC_Project2.git
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
traceTM_cander35.py	This Python script is designed to simulate the behavior of Non-deterministic Turing Machines (NTMs) or Deterministic Turing Machines (DTMs) based on a configuration provided in a CSV file. I wrote the script to trace the possible paths of computation for a given machine and input strings, while handling both deterministic and non-deterministic transitions seamlessly. The script reads the machine's states, symbols, transitions, and start/accept/reject states from the CSV file, ensuring robustness by ignoring extra columns and validating the expected format for each transition. For each input string, the script performs a breadth-first exploration of the machine's computation tree, tracking all configurations and

	stopping when an accept state is reached or when all paths lead to a reject state. It calculates key metrics, including the depth of the computation tree, the number of configurations explored, and the average nondeterminism (the degree of branching). I also integrated functionality to respect configurable limits on the maximum depth and transitions to prevent infinite loops or excessive computation. The results, including whether each string was accepted or rejected, transitions taken, and configurations explored, are logged into an output CSV file along with a formatted table for better readability. I created this tool to help me analyze and debug Turing Machines efficiently, especially when working with theoretical problems or automata coursework. It allows me to test multiple input strings in one go, while providing insights into the computational paths and behavior of the machine.
Test Files	
input_palindrome_cander35.csv	This file defines a Turing Machine designed to recognize palindromes over a specific alphabet. This machine's configuration includes a set of states, input symbols, tape symbols, transitions, and designated start, accept, and reject states. The transitions guide the machine to process the input string character by character, comparing symbols from both ends of the string to verify whether the input is a palindrome. It employs a combination of state changes, symbol rewrites, and head movements to achieve this, ensuring that it handles different lengths and compositions of strings systematically. I use this file as input to my Turing Machine simulator to test and analyze how the machine behaves when processing palindromic and non-palindromic strings. By running the machine with various inputs, I can observe its decision-making process, validate its correctness, and study its handling of edge cases like empty strings or single-character inputs. This file is particularly useful for

	---	---
		exploring fundamental concepts in automata theory and understanding how Turing Machines can be configured to solve specific problems.
	Output Files	
	output_results_cander35.csv	The output of the trace Python script, when passed the input_palindrome_cander35.csv file, provides a detailed summary of how the Turing Machine processes various input strings to determine whether they are palindromes. Each row in the output represents the results for a specific input string and includes details such as the depth of the computation tree, the total configurations explored, the transitions taken, and whether the string was accepted or rejected. The output also calculates the average nondeterminism, showing the machine's branching behavior during computation. It is organized into a table for readability. I use this output to analyze the performance and correctness of the palindrome-recognizing Turing Machine. It helps me understand how the machine processes different inputs, including valid palindromes, non-palindromes, and edge cases like empty strings. The detailed logs allow me to trace the paths explored by the machine, validate its transitions, and ensure it adheres to the expected computational behavior. This output is a valuable resource for debugging, optimizing, and gaining insights into the practical implementation of theoretical automata concepts.
	Plots (as needed)	
	n/a	n/a (the output file contains a table organizing the data)
8	Programming languages used, and associated libraries: Python - csv - os - argparse - collections - tabulate

9	Key data structures (for each sub-project): In my Python code, I rely on several key data structures to effectively simulate the behavior of a Turing Machine. The most important is the defaultdict from Python's collections module, which I use to store the machine's transitions. It maps each (state, symbol) pair to a list of possible transitions, allowing me to handle non-deterministic machines with ease. This structure ensures efficient access to all potential moves the machine can make from a given configuration. For breadth-first exploration, I use a deque to implement the computation queue. This double-ended queue allows me to append new configurations at the end and process them from the front, ensuring a systematic exploration of all possible paths. Each item in the deque is a tuple containing the current configuration (represented as a combination of the tape, state, and tape head position) and the path of configurations leading to it. To track explored configurations and avoid redundant processing, I use a set. This ensures that the machine doesn't revisit the same state-tape combination, saving computational resources and preventing infinite loops. Additionally, I use lists to store the computation tree and configurations explored at each depth level, enabling me to summarize the machine's behavior in a structured and clear manner. These data structures work together to provide an efficient and scalable way to simulate Turing Machines, handle non-determinism, and log detailed results for analysis. By combining them, I can focus on implementing the logic of the simulation while relying on Python's robust structures for efficient data management.
10	General operation of code (for each subproject): The general operation of my Python code begins with loading a Turing Machine's configuration from a CSV file, where I parse its states, symbols, transitions, and start/accept/reject states. Once the machine is initialized, I simulate its operation for a given set of input strings using a breadth-first search approach. Starting from the initial configuration, I systematically explore all possible paths the machine can take, using a queue to track configurations and a set to avoid revisiting previously explored states. For each configuration, I apply the machine's transitions, updating the tape, state, and head position accordingly, and add the resulting configurations to the queue for further exploration. If the machine reaches an accept state, I stop processing and record the path leading to acceptance. If all paths reach a reject state, I record rejection along with the maximum depth explored. To prevent infinite loops, I also respect limits on the maximum depth and number of transitions. Finally, the code summarizes the results for each input string, including whether it was accepted or rejected, the depth of the computation tree, the configurations explored, and the machine's average nondeterminism. These results are logged to an output CSV file, along with a formatted table for readability. This structured operation allows me to analyze how the machine processes different inputs and ensures that the simulation handles both deterministic and non-deterministic machines effectively.
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 input_palindrome_cander35.csv test case because it provides a

	straightforward yet effective way to test the correctness of my Turing Machine simulation. Palindrome recognition is a classic problem in automata theory, and using this test case allowed me to verify whether my code correctly handles a machine that checks if a string reads the same forwards and backwards. By running the test, I was able to observe how the machine processes different input strings, including palindromes, non-palindromes, and edge cases like empty strings. The results from the test case helped me assess the correctness of my code in several ways. First, it confirmed that the simulation correctly explores all possible computation paths, as seen in the detailed output of the configurations explored and the transitions taken. If the machine correctly identified a palindrome, the output showed that it reached the accept state, and if the string wasn't a palindrome, the machine correctly rejected it. Additionally, the output provided insights into how efficiently the machine explored the configurations, particularly in terms of depth and nondeterminism. Overall, the input_palindrome_cander35.csv test case validated that my code correctly simulates the behavior of a Turing Machine for a non-trivial computational problem, ensuring both accuracy and efficiency in the simulation process.
12	How you managed the code development: Throughout the development of the Python code, I focused on breaking down the problem into manageable components and ensuring that each part of the simulation was handled efficiently. I started by designing the core logic for loading the Turing Machine configuration from the CSV file, which required careful parsing of states, symbols, and transitions. Once that was in place, I implemented the breadth-first search algorithm to simulate the NTM's behavior, making sure to track configurations and handle non-deterministic transitions correctly. I then built functionality to log the results, including details on the machine's performance like transitions, depth, and nondeterminism. I iterated on the code by testing it with various inputs, starting with simple cases and gradually moving to more complex ones like the palindrome test. After identifying a few issues related to extra columns in the CSV file, I added error handling to cleanly manage these edge cases. I also focused on organizing the output into both a raw CSV and a formatted table to make it more readable. Throughout the process, I kept testing and refining the code, ensuring it was robust, efficient, and met the project requirements. Managing the development in small, testable chunks helped me ensure each part of the code was working correctly before moving on to the next, leading to a more stable final implementation.
13	Detailed discussion of results: When using the input_palindrome_cander35.csv test file with my Python code, the results provided detailed insights into how the Turing Machine processes various input strings to determine if they are palindromes. For palindromic strings like abccba or racecar, the machine correctly reached the accept state (q_{acc}), and the output showed the depth of the computation tree, the number of configurations explored, and the transitions taken. The average nondeterminism was also calculated, showing how many branching paths were considered during the computation. For non-palindromic strings like abcd or abc, the machine reached the reject state (q_{rej}), and the output reflected the maximum depth reached before rejection, along with the configurations explored and transitions considered.

	The test also helped me verify how the machine handles edge cases such as empty strings or single-character strings. For these cases, the machine immediately accepted the string, which was reflected in the output showing minimal transitions and depth. The formatted table generated by the script allowed me to quickly interpret the results, presenting a clear summary of the machine's performance for each input string. Overall, the results from the palindrome test case confirmed that my code was correctly simulating the Turing Machine's behavior, handling both palindromes and non-palindromes as expected. The detailed output, including the configurations explored, transitions taken, and performance metrics like depth and nondeterminism, provided me with valuable insights into the machine's decision-making process, ensuring that my simulation was both accurate and efficient.
14	How the team was organized: I worked alone, so I was responsible for all the work.
15	What you might do differently if you did the project again: I think my main issue was the fact that I began writing my NTM code by using Deterministic input files from the shared Google Drive. I wish I would've started with non-deterministic inputs because it would have been able to deal with deterministic inputs without more intervention.

16	Any additional material:
	How to run my code-
	There are several ways to run my code depending on the input parameters you want to specify and the desired configuration. Below, I'll outline the different ways you can run the code from the command line:
	1. Basic Run with Default Settings
	
	If you just want to run the script with the default input strings and without modifying the maximum depth or transition limits, you can simply provide the machine CSV file and an output file. This will use the default values for the inputs (aaa, 111000, abc, and "") and limits (max_depth=1000, max_transitions=1000).
	
	`python3 trace_script.py input.csv output_results.csv`
	
	(for the files in my repo: `python3 traceTM_cander35.py input_palindrome_cander35.csv output_results_cander35.csv`)
	
	4. Run with Flags
	
	You can combine multiple options in one command to run the script with custom inputs, custom maximum depth, and custom transition limits. This allows you to fine-tune the simulation based on your specific requirements.
	
	Example using all flags, but don't need all because there are defaults:
	
	`python3 traceTM_cander35.py input_palindrome_cander35.csv output_results_cander35.csv --inputs abccba racecar --max_depth 500`

	--max_transitions 2000 ● --inputs abccba racecar: Specifies custom input strings to test the Turing Machine. ● --max_depth 500: Sets a limit on the computation tree depth. ● --max_transitions 2000: Limits the total number of transitions simulated.