Project 2 Readme Team hberens

1	Team Name: hberens
2	Team members names and netids: Helena Berens, hberens
3	Overall project attempted, with sub-projects: Tracing NTM behavior- given a TM in the form of a CSV file, an input string, and a max_depth limit, I trace through the transitions of the TM, figuring out if the string should be accepted or rejected, and then print out the configurations of an accepted string.
4	Overall success of the project: This project was successful as it successfully and correctly simulates the functions of a Turing machine upon reading TM data from a file, properly applying transitions based on the machine's state and user-inputted string. It handles any input string the user types in and determines whether a string is accepted or rejected by tracing the machine's execution. I know that it works successfully because I tested the code on various Turing Machines in different CSV files that we constructed throughout the semester, and inputted strings for which I knew the output I desired.
5	Approximately total time (in hours) to complete: ~20 hours
6	Link to github repository: https://github.com/hberens/theory_project02_hberens - on the project03 branch
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_hberens.py	This file contains all the code that reads in the CSV file for a Turing machine, defines classes for the TM and the tape, and asks the user for input pertaining to what TM they want to use and what string they want to run. It has functions append_transitions and trace_string that read through the transition and rewrite any characters and move the string and head accordingly, and then also test transitions for a given string until either accepting or rejecting, and writing various pieces of information to an output file.

Test Files	
0^(2^(n))_hberens.CSV aplus_hberens.CSV a*b*c*_DTM_hberens.CSV even_a_hberens.CSV a*b*c*_NTM_hberens.CSV w#w_hberens.CSV	These files are all CSV files representing various Turing machines that are described in the name of the file. They all contain information about the Turing machines in the format described in the project description. They contain the name of the machine, states, sigma, gamma, start state, accept state, reject state, and then all of the transitions.
Output Files	
output-0^(2^(n))_hberens.txt output-aplus_hberens.txt output-a*b*c*_DTM_hberens.txt output-even_a_hberens.txt output-a*b*c*_NTM_hberens.txt output-w#w_hberens.txt	These output files are the text output files produced when the user inputs any of the CSV files seen above as the TM they want to use. The names align respectively. They contain the following information for every string the user inputs: machine name, string inputted, the depth of the tree configuration, total transitions take, whether the string was accepted or rejected and in how many transitions, those transitions if the string was accepted, or if the max depth was surpassed.
Plots (as needed)	
N/A	

8	Programming languages used, and associated libraries: Python; CSV, collections, os
9	Key data structures (for each sub-project): 1) **Dictionaries** were central to the implementation of the transition table, allowing for efficient mapping of the current state and tape symbol to the next action (state, symbol, head direction). - The primary use is in the TM_Tape dictionary, which stores the transition rules for the Non-Deterministic Turing Machine (NTM). This dictionary is structured with the key being the current state of the Turing Machine, and the values being another dictionary. This value dictionary has a key that is the read/input symbol on the tape wherever the head is, and the value is a list of tuples where each tuple contains (the next state, the symbol of write, and the direction to move the head) for each possible transition for that state and input character. - This structure allows efficient lookup of possible transitions based on the

    - current state and the symbol being read by the Turing machine's head. When a state and symbol are encountered, you can quickly find the corresponding transitions and apply the necessary state changes, symbol writes, and head movements.
    - In the append_transitions method, the dictionary is used to retrieve all possible transitions based on the current state and the symbol under the tape head. The method checks if the current state and the symbol are valid keys and, if so, iterates over the possible transitions. This approach allows you to handle nondeterministic transitions, where multiple valid next states may exist for a given input, by storing them in a list within the dictionary.
- In the trace_string method, a dictionary called visited is used to track the states that have been visited at each depth level of the simulation.
    - The keys of this dictionary are the depth levels, and the values are lists of tape configurations at that level. This provides a way to track the progress of the simulation, including when a state is accepted or rejected.
    - By storing these configurations at each level, it helps for printing out the configurations of accepted strings later. This dictionary also helps with limiting the depth of the search based on the max_depth given by the user.
- Lastly, in the trace_string method dictionaries transitions_per_level and non_leaves_per_level were used
    - These dictionaries had keys for each level, and the values were integer values representing the number of outgoing edges at each level of the tree and the non-leaf nodes at each level, respectively. This data is used to calculate the degree of nondeterminism at the end of the string trace.

2) **Lists/Arrays** were used within the dictionaries as described above and for managing configurations in nondeterministic machines.

- Strlist: a list was used to represent the user-inputted string we want to trace in order to iterate over it and create a tape based on it.
- In the Tape class, lists are used to represent the left and right sections of the Turing machine's tape.
    - The tape is divided into three main components: Left: A list representing the symbols to the left of the current tape head. Right: A list representing the symbols to the right of the current tape head. Head: A single symbol representing the current symbol under the tape head.
    - These lists allow for the manipulation of the tape, such as adding new symbols or moving the head to different positions.
- In the trace_string method, lists are used to maintain the path of execution through the Turing machine. The path consists of a sequence of tape configurations (represented by the Tape class), where each configuration represents a step or state during the Turing machine's computation.

3) **Tuples and Sets** were used for immutable keys in dictionaries, ensuring the state-symbol mappings didn't change as well as being used for states input symbols, and the tape alphabet.

- The states of the Turing machine (self.states), the input symbols that the machine can read (self.sigma), and the gamma symbols that can appear on the tape (self.gamma) are stored as sets to ensure uniqueness and facilitate fast membership checking.
- Tuples were used to store transition in the TM_Tape dictionary as described above to ensure they were immutable.

4) **Queue/Deque** is used in the trace_string method to store and manage the configurations of the Turing machine (represented by instances of the Tape class) as they evolve during the execution process.

- The deque is initialized with the starting configuration, which consists of the initial state, the input string, and the position of the tape head. q = deque([(tape, 0, [tape])])
    - tape: This represents the current tape configuration, which includes the current state, the left portion of the tape, the head, and the right portion of the tape.
    - 0: This represents the current depth or level in the computation, starting at 0.
    - [tape]: This is a list that stores the path of configurations taken to reach this state (initially just the starting configuration).
- At each step of the process, the algorithm pops the first element from the front of the deque, processes the current state, and generates new configurations based on the available transitions. The new configurations are then appended to the back of the deque to continue the search- current, level, path = q.popleft()
- The deque allows the program to explore all possible configurations of the machine in a breadth-first manner, ensuring that all transitions are explored before moving deeper into the execution. This is important for nondeterministic Turing machines, where multiple possible transitions can be made from a given state and input symbol.
- Using a deque ensures that both enqueueing and dequeueing operations are performed in constant time, O(1), which is important for maintaining efficient performance

5) **Classes** are used to represent both the Tape and the Turing Machine itself

- The Tape class is used to represent the configuration of the Turing machine's tape at any given moment during the computation. A Turing machine operates on an infinite tape that holds the input string, along with the tape head that moves left or right while reading and writing symbols. In this context, the Tape class models the tape configuration using both its __init__ and __str__ methods.

	- curr_state: Stores the current state of the Turing machine. - left: Represents the left part of the tape, which is an array of symbols to the left of the head. - head: Represents the current symbol under the tape head. - right: Represents the right part of the tape, which is an array of symbols to the right of the head. - The NonDeterministicTuringMachine class models the entire NTM, including its state transitions, input alphabet, tape, and behavior. This class is responsible for loading the machine's definition from a CSV file, storing its components, and running the simulation of the Turing machine on an input string using the __init__, append_transitions, and trace_string methods. - name: The name of the Turing machine - states: A set of all possible states of the machine - sigma: The input alphabet - gamma: The full set of symbols the Turing machine can write - start_state: The initial state of the machine. - accept_state: The state that indicates the input string is accepted. - reject_state: The state that indicates the input string is rejected. - TM_Tape: A dictionary mapping each state to possible transitions based on input symbols
10	General operation of code (for each subproject): **1. __init__ in Tape Class:** - This is the constructor for the Tape class, which is used to represent the tape of the Turing machine. It initializes four attributes: curr_state (the current state of the machine), left (the tape to the left of the head), head (the current symbol under the machine's read-write head), and right (the tape to the right of the head). These attributes are set when a new Tape object is created, with default values for left, head, and right if not provided. **2. __str__ in Tape Class:** - This function defines how to represent the Tape object as a string. When a Tape object is printed, it returns a string that shows the left part of the tape, the curr_state, the head symbol, and the right part of the tape. **3. __init__ in NonDeterministicTuringMachine Class:** - The constructor for the NonDeterministicTuringMachine class initializes the machine based on data from a CSV file. It takes the file path as an argument and reads the CSV file using the CSV module. It parses the machine's properties such as the name, states, input alphabet (sigma), tape alphabet (gamma), start state, accept state, and reject state. Additionally, it constructs a dictionary (TM_Tape) for the machine's transitions. **4. append_transitions in NonDeterministicTuringMachine Class:** - This function computes the next possible tape configurations based on the current configuration and the transition rules of the Turing machine. It takes a

Tape object as input and checks for available transitions in the machine's current state and under the current symbol under the tape head.
- If transitions exist, it creates new Tape objects for each possible transition, applying the appropriate head movement (R for right or L for left) and updating the tape accordingly. The function returns a list of new Tape configurations that result from applying all the transitions.

5. trace_string in NonDeterministicTuringMachine Class:
- The trace_string function simulates the execution of the Turing machine on a given input string up to a specified depth. It initializes the tape with the input string, with the head positioned at the first character.
- The function uses a breadth-first search (BFS) approach, maintaining a deque (q) to explore possible configurations at each step. It tracks visited configurations in a visited dictionary. The function continues processing until it reaches the maximum depth or either the accept or reject state.
- It also initializes 2 dictionaries to track the outgoing transitions per level and the number of non-leaf nodes and updates them at each level in the while loop. At the very end, it calculates the degree of nondeterminism using the calculation given in the instructions and prints this data. It also prints other information such as configuration depth and total transitions taken.
- If the machine accepts the string, it prints the sequence of transitions leading to acceptance; if the string is rejected, it prints the number of transitions taken. The results are printed to the terminal and written to an output file.

6. main Function:
- The main function starts by prompting the user to input the CSV file for the Turing machine, checks if the file exists and reloads the file prompt as necessary. It then creates a Turing machine using the class from the CSV file, and it creates an output file to log the results of running strings through the machine. It enters a loop, asking the user for input strings to simulate on the Turing machine. For each string, the user is asked for a maximum depth limit, and the trace_string function is called to simulate the machine. The results are printed to the console and written to the output file. The loop continues until the user enters the quit command.

11 What test cases you used/added, why you used them, what did they tell you about the correctness of your code.
I used 6 different test Turing Machine CSV files to test my code. I used the aplus one given to us, even_a, a*b*c* in deterministic form, and a*b*c* in nondeterministic form which were all given by other students to Professor Kogge. I made 2 more based on example machines I had drawn from the textbook called 0^(2^(n)) and w#w.
I used all of these because I had then drawn on paper, and could easily trace strings that I attempted to input into my code. I also used them because they varied in size and description, and so I thought the variance helped ensure I managed all the edge cases that came up with these TMs.

	These test cases indicated my code at least ran by successfully being read in and producing output files for each TM. They also helped me check that my code not only ran, but worked as required, since I traced many strings by hand to figure out the tree depth and the various configurations that led to accept or reject states. Thus, when my code printed out the same results I got by hand, I was confident it worked. Additionally, having the same machine but in DTM and NTM form helped me confirm that my code worked both deterministically and nondeterministically while confirming that the total transitions could be more than the transitions to an accept state with the NTM. Furthermore, since I had both DTM and NTM example machines, I was able to test that my calculation for the degree of nondeterminism was correct because I got either 1 or 0 for all of the DTM machine string inputs depending on whether they were rejected right away or not, and I got double values greater than or equal to 1 for the NTM machines like the a*b*c*_NTM and aplus ones.
12	How you managed the code development There were several stages in my process of code development as outlined below. 1) First I had to understand the problem at hand. I read the textbook to fully understand how NTM's worked and to understand the structure and behavior. I drew multiple examples and traced through many strings to later check my code using these. 2) Second, I started designing classes for the Tape and the NTM. These 2 classes were essential in structuring how I traced through the strings. The Tape class was relatively small and only needed the state, head, and direction, along with an __str__ method for printing configurations. I planned the NTM class structure to initialize a bunch of attributed and parse through the CSV file along with the transitions in __init__, and then use other methods to read transitions and update the tape, trace through a string and track the machine's execution, and print out output. 3) Thirdly, I implemented all of the methods for the NTM class, slowly debugging parsing the csv file and constructing the dictionary of transitions. I then made the append_transition method which was also relatively small in order to go through the transitions possible given a certain state and input symbol, look at each one, update the head based on the direction indicated, and create new tapes representing the results of each transition. Then I needed a way to trace an input string, choose the best transition upon calling the append_transitions method, and print out the configurations leading to an accept state. To do this I created the trace_string method that creates a Tape given a specific input string and traces through possible transitions using a BFS-like approach, adding them to a visited dictionary as it goes. The full implementation of this method is described earlier. 4) The last step I took in coding the implementation was to create the main function to ask the user for input for the TM, string, and max depth, use these to create classes, and then call the appropriate methods to trace the string. 5) After coding the implementation of the Turing machine, I went through and debugged any errors I was getting when trying to run the file along with any errors I was getting in the output. I added print statements throughout to ensure

each individual portion was reading in data and created variables as I desired. I also went through and added a lot of error checking, such as with the user input, and also added cases for certain edge cases like the empty string.

6) The last step of my process was to ensure all my output was correct by making more test Turing Machine files and running each and every one of them multiple times with different strings and comparing the output to what I got by hand. I updated any error messages and printing lines to be better structured and clear.

13 Detailed discussion of results:

- As discussed above, I used 6 different input Turing Machine files to test. They all ran successfully even when typed with extra whitespace. To run the program you just do ./traceTM_hberens.py and then input the desired Turing machine CSV file from the list of files given at the beginning of the document. Then input any string you want to run through it as well as the max depth you have to stop at.
- The code works completely, and will create an output file called output-{the csv file name the user gave}.txt which will include the name of the machine, inputted string tested, the depth the string went through the tree configuration, the total number of transitions taken throughout tracing, the degree of nondeterminism, the number of transitions until either acceptance or rejection / if tracing was stopped after the max depth was reached, and a list of the configurations to acceptance if the string was accepted.
- The results show a successful trace of any string given with the correct depth level reached and number of transitions traversed as well as the correct calculation for the degree of nondeterminism. In the output files, I have provided at least 2 accepted strings, at least 1 rejected string, at least 1 string that's execution was stopped because it reached the max depth, and the empty string to check an edge case.
- Here is a table with some of the example strings I ran through all 6 test files:

Name of Machine	String	Accepted or Rejected	Depth	Configurations Explored	Average Non-Determinism
aplus	aaa	Accepted	4	4	1.75
aplus	aaaaaaaaaaa	Execution stopped	4	5	2.00
even_a	aaa	Rejected	3	3	1.00
0^(2^(n))	0000	Accepted	21	21	1.00
a*b*c*_NTM	abc	Accepted	4	7	2.29
a*b*c*_NTM	aaabbcccc	Accepted	10	17	2.47
a*b*c*_DTM	aaabbcccc	Accepted	10	10	1.00

w#w	100#01	Rejected	4	4	1.00
w#w	#	Accepted	2	2	1.00

Average nondeterminism measures how many different transitions at each state and then averaged over the whole execution of the machine for a given input string. From the table we can see that for the deterministic Turing Machine, the average non-determinism was 1. We can also see that the longer the string, the more depth has to occur, so there's a larger chance for a high average nondeterminism. Generally speaking, there was pretty low nondeterminism as everything stayed below 3 for most of my inputs. For inputs that were rejected, there was low non-determinism if they reached rejection very quickly. For strings that stopped execution because they reached the max depth the user inputted, the average nondeterminism is not very meaningful.

14	How team was organized I completed this project by myself, so I did all of the coding.
15	What you might do differently if you did the project again If I were to do the project again, I would break down large classes and functions into smaller, more manageable components. For instance, I would separate the tape management and state transitions into distinct classes, which would make the code easier to maintain and extend. To improve user interaction, I would provide more structured guidance for inputs, possibly through prompts that offer specific options for how the Turing machine should behave. While the dictionary-based transition system is functional, I would explore more efficient data structures, for faster lookup and transition handling. Furthermore, I would consider using parallelism to explore many paths simultaneously, reducing the time required for complex Turing machine simulations.
16	Any additional material: None