Project 2 NTM Tracing Readme

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"

Machine Name	**Input string**	**Result**	**Depth**	**# of Configurations Explored**	**Average non-determinism**
a_plus_mzitella.csv	aaa	Accept	4	11	1.14
a_plus_mzitella.csv	aaaaa	Accept	6	17	1.27
a_plus_mzitella.csv	z	Reject	1	2	1.00
abc_star_mzitella.csv	aabbbc	Accept	7	50	1.62
abc_star_mzitella.csv	cbcbcb	Accept	2	5	1.33

1	Team Name: mzitella
2	Madeline zitella Net ID: mzitella
3	Overall project attempted: - Given a NTM and a string, create a program to trace all possible paths the NTM might have taken and stop at an accept, reject, or when the maximum depth is reached. (i.e. Program 1 from project instructions) Sub-projects: - Parse the NTM .csv file, including the header information and state transitions - Get all the possible configurations at each level and return as a list. - Use Breadth_First Search to explore every possible path and maintain a tree of configurations - Calculate nondeterminism - Print all output specified in instructions
4	Overall success of the project:

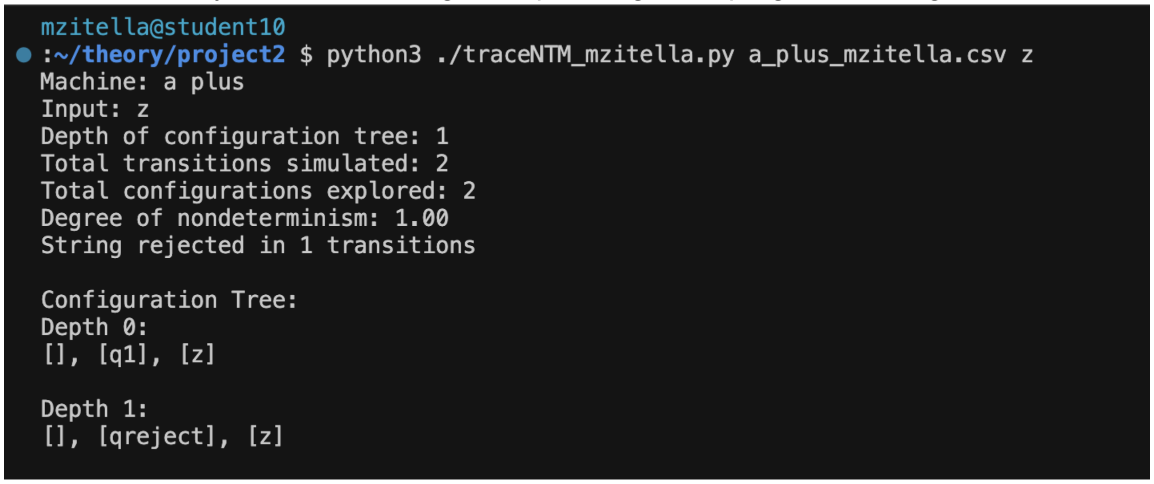
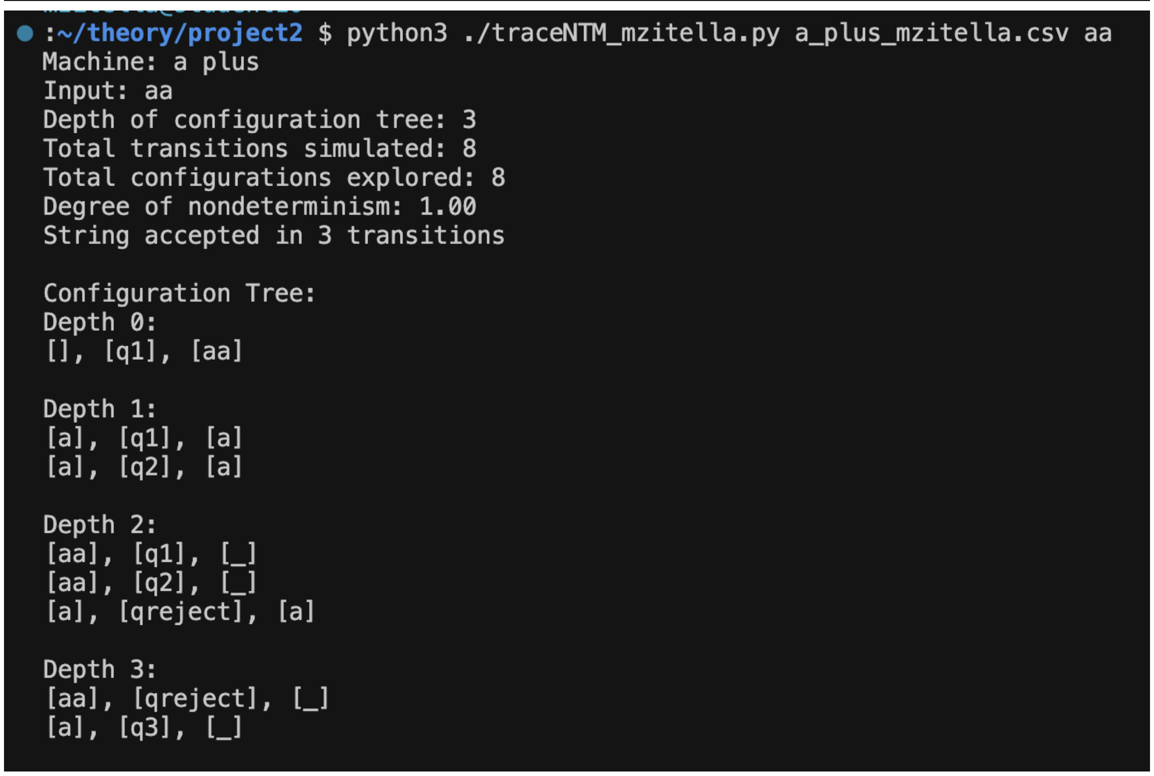
	I created a NTM trace program that successfully ran on all NTM input csv files, which helped justify that my algorithm, use of data structures, and method to trace the NTM was correct. The project helped me better visualize how NTM's can be visualized and efficiently traversed through tree structures.
5	Approximately total time (in hours) to complete: ~5 hours
6	Link to github repository: https://github.com/MadelineZitella/Theory-of-Computing-NTM-Tracing-Project
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_mzitella.py	This is the main program file that parses the .csv TM file, handles tape movement by exploring all possible configurations using BFS, and returns the accepting path if there is one. All output is printed to the terminal. To use this file, run: `python3 ./traceNTM_mzitella.py <tm_file> <input_string>`
Test Files	
abc_star_mzitella.csv	NTM for a*b*c* formatted as a csv file.
a_plus_mzitella.csv	NTM for a^+ formatted as a csv file.
Output Files	
abc_star_output_example_mzitella.txt	Output for program running on abc_star_mzitella.csv

	with the string aaaaabcccccc
a_plus_output_example_mzitella.txt	Output for program running on a_plus_mzitella.csv with the string aaaaaaa

8	Programming languages used, and associated libraries: - Python - Python csv module used to parse csv input files - Python sys module used to take in arguments (i.e. the csv file describing the machine and the input string to run - Python typing module used to make sure function outputs are of the right types as intended
9	Key data structures (for each sub-project): The first ket data structure is a dictionary named “transitions” that stores information about each transition for the NTM including the current state and symbol as keys, with the new state, new symbol, and direction as values. Next, in the get_next_configs() function, I used a tuple to store the current configuration of the NTM at a single step, including the left side, current state, and right side. This tuple is named “config.” In the BFS algorithm inside of the trace() function, I used a list of lists of tuples in order to manage all configurations of the NTM at each depth level. This list of lists is named “config_tree.” Each list inside of the larger list represents an individual configuration. Also in the trace() function I used a list of lists named “path” in order to track the path from start to accept state. In order to store the next set of configurations to explore, I used another list of lists named “next_level.” It is important to note that for Breadth-First search algorithms we operate as if we are processing a queue in order to ensure that we explore the tree laterally and get the shortest path.
10	General operation of code (for each subproject) - Parse the NTM .csv file, including the header information and state transitions - Using python’s csv module and a dictionary helped me parse csv file input in the __init__ and parse_ntm methods. The transitions dictionary maps each current state and symbol to their respective possible transitions. - Get all the possible configurations at each level and return as a list. - In the get_next_configs() method, all of the possible next configurations for a given configuration are computed. Starting at the symbol after the head, the next configuration is updated, including left, new_state, and right. The next_configs array is appended to a larger list to hold every configuration. - Use Breadth_First Search to explore every possible path and maintain a tree of configurations - The trace() function traces the NTM on a given input using BFS. Using a

	max depth of 100 and an input string, we first start the configuration using "config_tree." For each level of the tree, we explore all current configurations by calling get_next_configs. If the configuration reaches q_{accept} the machine accepts and the accepted path is extracted by calling the extract_accepting_path() function. However, if we run out of configurations to explore before an accept state is reached, we reject by checking if "next_level" is empty. - Calculate nondeterminism - Also in the trace() function we calculate the degree of nondeterminism. The instructions define this measurement as the average number of new configurations that come from an average configuration, with a 1 corresponding to a deterministic computation. Upon the accept state being reached, I calculate nondeterminism by dividing the total transitions by the non-leaf configurations. We want to divide by the number of non-lead configurations because we only want to consider the configurations that generate at least one new configuration. This can help indicate to us how many possible configurations can be reached from any given configuration. - Print all output specified in instructions - In main() the program prints out the machine name, the given input string, depth of the tree, number of transitions, degree of nondeterminism, the number of transitions before the string was accepted, rejected, or the program was stopped, and the whole configuration tree, showing how the states were traversed.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. I developed my code initially by using the a^+ csv file described in the project instructions. Having a specific example of state and transitions to reference while building and testing the program helped me. I used this file because the instructions give an idea of what reasonable output may look like for this machine. Once I verified my output for a^+, I moved on to the other test files supplied on canvas in order to de-bug and check that the degree of determinism and BFS algorithm was working properly. Included above under the files is the abc_star_mzitella.csv input file, for example.
12	How you managed the code development I first thought at a high-level about how I wanted to store the data, how to create a tree structure, and what BFS would look like. After writing some pseudocode and completing the functions to parse the csv input, I moved on to incrementally writing functions and using print statements to debug as I went. As I mentioned before, having an expectation of what the a^+ machine should output helped verify that my program was functioning as intended.
13	Detailed discussion of results: After running my program on several machines, it is evident that the degree of nondeterminism reflects the average number of choices available at each step of configurations. For some machines, like a+, there is a low degree of nondeterminism because there are always 2 choices at every step. This means that even with longer input strings, the level of non-determinism doesn't grow very quickly. However with a*b*c*, there is a higher degree of nondeterminism because it has more choices at each

	level of configuration; there are more transitions possible per non-leaf node configuration on average.
14	How team was organized N/A (worked individually)
15	What you might do differently if you did the project again: If I were to do this project again, I would review more thoroughly how to structure data for a BFS before I began to write code. I think this would have prevented many of the bugs I ran into and saved time so that I didn't have to stop and review.
16	Any additional material: Screenshot of my terminal showing example usage and program running:  

```
mzitella@student10
● :~/theory/project2 $ python3 ./traceNTM_mzitella.py a_plus_mzitella.csv z
Machine: a plus
Input: z
Depth of configuration tree: 1
Total transitions simulated: 2
Total configurations explored: 2
Degree of nondeterminism: 1.00
String rejected in 1 transitions

Configuration Tree:
Depth 0:
[], [q1], [z]

Depth 1:
[], [qreject], [z]
```

```
● :~/theory/project2 $ python3 ./traceNTM_mzitella.py a_plus_mzitella.csv aa
Machine: a plus
Input: aa
Depth of configuration tree: 3
Total transitions simulated: 8
Total configurations explored: 8
Degree of nondeterminism: 1.00
String accepted in 3 transitions

Configuration Tree:
Depth 0:
[], [q1], [aa]

Depth 1:
[a], [q1], [a]
[a], [q2], [a]

Depth 2:
[aa], [q1], [_]
[aa], [q2], [_]
[a], [qreject], [a]

Depth 3:
[aa], [qreject], [_]
[a], [q3], [_]
```