Project 2 Readme Team crysalball

Version 1 9/11/24

1	Team Name: **crystalball**
2	Team members names and netids: **Jack O'Connor (joconn25), Joseph Bertram (jbertra3), Brian Rabideau (brabidea)**
3	Overall project attempted, with sub-projects: **Program 1: Tracing NTM Behavior** **Program 2: K-Tape Turing Machine** **Program 3: Other… 2 Stack Turing Machine**
4	Overall success of the project: **Successful**
5	Approximately total time (in hours) to complete: **10**
6	Link to github repository: https://github.com/Jacko7973/TOC-Project-2-crystalball/
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_crystalball/TM_utils_crystalball.py	Utility functions for the NTM tracing program
./traceTM_crystalball/traceTM_crystallball.py	Main executable for NTM implementation
./traceTM_crystalball/data_equal_01s_crystalball.csv	NTM CSV for testing
./traceTM_crystalball/data_aplus_crystalball.csv	^
./traceTM_crystalball/data_loop_crystalball.csv	^
./traceTM_crystalball/data_equal_01s_DTM_crystalball.csv	^
./ktape_crystalball/ktape_crystalball.py	Utility functions for the k-tape DTM program
./ktape_crystalball/data_flip_crystalball.csv	k-tape DTM CSV for testing
./ktape_crystalball/data_left_crystalball.csv	^
./ktape_crystalball/data_right_crystalball.csv	^

<table>
<tr><td></td><td>
<table>
<tr><td>./ktape_crystalball/data_multiple1_crystalball.csv
./ktape_crystalball/data_multiple2_crystalball.csv

./program3/prog_3_crystalball.py</td><td>^
^

Main file for two stack TM</td></tr>
<tr><td colspan="2">Test Files</td></tr>
<tr><td>./traceTM_crystalball/test_traceTM_crystalball.py

./ktape_crystalball/test_ktape_crystalball.py

./program3/data_transitions_crystalball.csv
./program3/prog_3_test_crystalball.py</td><td>Test script for NTM implementation

Test script for k-tape DTM implementation

Transition input file
Test script</td></tr>
<tr><td colspan="2">Output Files</td></tr>
<tr><td>./traceTM_crystalball/output_results_crystalball.csv

./ktape_crystalball/output_result_crystalball.csv

./program3/output_program_3_crystalball.csv</td><td>Output table for test script

Output table for test script

Output file</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td></td><td></td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries:
Programming Languages: Python
External Libraries: N/A</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):
Program 1: Tracing NTM Behavior
- List
- Queue
Program 2: K-Tape Turing Machine
- List
Program 3: Other… 2 Stack Turing Machine
- queue
- Dictionaries
- Sets
- list</td></tr>
</table>

10	General operation of code (for each subproject) **Program 1: Tracing NTM Behavior** - **The majority of the logic used to simulate the Nondeterministic Turing Machine is located in the TM_utils_crystalball.py file. Within that, namedtuples are declared to provide a framework for managing Turing Machines and their operations. The load_TM() function converts a CSV turing machine into a TuringMachine object in the code. Then, the init_TM() function is used to set up a starting state for a given turing machine and input string. Finally, the trace_TM function takes that state, and traces the steps the nondeterministic truing machine will take while processing that string. The outputs of this function include whether or not the machine accepted the string, how many states were explored, the maximum depth of the tree, and the average nondeterminism produced by the TM.** - **The traceTM_crystalball provides a command-line wrapper for running the trace on a given turing machine file and input string. It takes the filename of the CSV turing machine, the input string, and the maximum depth to explore to as command line arguments. It then runs the trace, outputting statistics of the trace, and if the machine accepts the string, it shows the path taken to accept the string.** **Program 2: K-Tape Turing Machine** - **The majority of the logic used to simulate a k-tape Deterministic Turing Machine is located in ktape_crystalball.py. The process_csv() function just loads the machine file and parses it based on what a normal machine file expects (machine name, number of tapes, states, sigma, gamma, start, accept, reject, and the transitions). The process_transitions() function processes the transitions and groups the old characters, new characters, and the direction of where the tape head moves. Internally, the function calls bool_move() function, which checks whether the transition is valid by checking if the current state matches the transition's current state and if the current character in the tape matches the current character in tape under the head specified by the transition. If the bool_move() function returns true, it then proceeds to call the replace_characters() function and the move_tape() function. The replace_characters() function just replaces the current character in the tape under the head with the new character specified by the transition. The move_tape() function just moves the head either left, right, or stay specified by the transition.** **Program 3: Other… 2 Stack Turing Machine** - **The main operation of the function involves two stacks of the TM which the string is initially pushed onto the first one then as the characters reach the top they are processed and switched to the second (where relevant). It keeps track of the state and prints out a representation of the two stacks each time. I made a transition dictionary with the current state and two stack symbols and it maps to the next state and what is pushed to the stack. It increments step to keep track of depth. It can read from csv file with the csv parsing function at the top and the commented code.**
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. **Program 1: Tracing NTM Behavior** - **I used four different Turing Machines to test the output of the NTM**

	program. For each of these turing machines, I ran the simulation on a number of input strings, some of which were to be accepted by the machine and some were to be rejected. On the aplus machine for example, I gave it the string "aaaaa" which it should accept, and "aaba" which it should reject. I repeated this with each of the four machines I tested. I even included a machine that intentionally loops forever to test the limiting system. **Program 2: K-Tape Turing Machine** **- I used three different Turing Machines to test the output of k-tape DTM program. The first machine flips the character of the input string (ex. If the input string is aa, the new string should be bb_), the second machine does the right shift operation (ex. If the input string is aa, the new string should be $aa), while the third machine does a combination of both. For each of these machines, I ran the simulation on a number of input strings (like aa, bb, aabb, bbaa). I made the machines simple, so I just created the machine with the appropriate transitions based on the input strings feeded in, so all test cases should end up being accepted.** **Program 3: Other… 2 Stack Turing Machine** **- My script utilizes the regex a+ and then runs unit tests on a number of strings that are either accepted or rejected. I have also run it with 10 other regex expressions but omitted them from the code to avoid having the code be too long. Also the commented code can read from a csv file set the required fields and determine if the string passes or not.**
12	How you managed the code development **We split this project up so each person implemented one of the programs independently. To manage the development environment, we set up a git repository and uploaded it to GitHub for collaboration.** **We chose to write the code in Python for simplicity and flexibility.**
13	Detailed discussion of results: **Program 1: Tracing NTM Behavior**

	Machine Name	Input String	Result	Max Depth	# Transitions	Average Nondeterminism	Comments
2	data_aplus_crystalball.csv	aaaaa	Accpeted	7	16	1.455	
3	data_aplus_crystalball.csv	aaaab	Rejected	6	13	1.444	
4	data_aplus_crystalball.csv	aaaaaaaaaaaaaaaaa	Accpeted	20	55	1.486	
5	data_aplus_crystalball.csv	baa	Rejected	2	2	1.0	
6	data_aplus_crystalball.csv	a	Accpeted	3	4	1.333	
7	data_aplus_crystalball.csv	aaacaa	Rejected	5	10	1.429	
8	data_equal_01s_DTM_crystalball.csv	01	Accpeted	6	6	1.0	
9	data_equal_01s_DTM_crystalball.csv	1010	Accpeted	14	14	1.0	
10	data_equal_01s_DTM_crystalball.csv	10101	Rejected	15	15	1.0	
11	data_equal_01s_DTM_crystalball.csv	11111111110000000000	Accpeted	312	312	1.0	
12	data_equal_01s_crystalball.csv	01	Accpeted	6	9	1.125	
13	data_equal_01s_crystalball.csv	1010	Accpeted	14	22	1.1	
14	data_equal_01s_crystalball.csv	10101	Rejected	15	23	1.1	
15	data_equal_01s_crystalball.csv	11111111110000000000	Accpeted	312	387	1.027	
16	data_loop_crystalball.csv		Timed Out	1000	1000	1.0	
17	data_loop_crystalball.csv	a	Timed Out	1000	1000	1.0	
18	data_loop_crystalball.csv	aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	Timed Out	1000	1000	1.0	

This table represents the output of each of the 17 tests run for the NTM component of this project. The first machine I tested it on was the aplus machine, which accepts the regular expression "a+". This machine, as expected, accepts all input strings with one or more a's and no other letters. The max depth column represents the deepest level of the search tree that the NTM explored before finding a solution. The # Transitions column represents how many different configurations were computed before finding a solution. Since this machine is nondeterministic, you can see that it posts values >1 for the average nondeterminism column, meaning at certain points, the machine has multiple options for a transition. The same can be seen, as expected, for the equal_01s machine, which also has nondeterministic transitions. The equal_01s_DTM and loop machines are both deterministic, so they only display values of 1 for the average determinism. Additionally, the table shows that the simulation is stopped after 1000 levels of search with the loop machine, which intentionally loops infinitely. Each machine accepts and rejects the strings as expected, proving that the simulation is being performed correctly.

Program 2: K-Tape Turing Machine

	Machine Name	Tapes	Input String	Result	Final State	Transitions
2	flip	aa_	1	bb_	qaccept	3
3	flip	aabb_	1	bbaa_	qaccept	5
4	flip	bb_	1	aa_	qaccept	3
5	flip	bbaa_	1	aabb_	qaccept	5
6	right shift	aa_, ____	2	aa_, $aa_	qaccept	4
7	right shift	aabb_, _____	2	aabb_, $aabb_	qaccept	6
8	right shift	bb_, ____	2	bb_, $bb_	qaccept	4
9	right shift	bbaa_, _____	2	bbaa_, $bbaa_	qaccept	6
10	flip and right shift	aa_, ___, ____	3	aa_, bb_, $aa_	qaccept	4
11	flip and right shift	aabb_, _____, _____	3	aabb_, bbaa_, $aabb_	qaccept	6
12	flip and right shift	bb_, ___, ____	3	bb_, aa_, $bb_	qaccept	4
13	flip and right shift	bbaa_, _____, _____	3	bbaa_, aabb_, $bbaa_	qaccept	6

The table outputs 12 different simulations from three different machines (flip, right shift, flip and right shift). The three machines each use one, two, and three tapes. The number of tapes chosen for the three machines were based on the intuitive sense of what number of tapes are best to solve the problem. For each machine, I ran a simulation of four trials. The number of input strings feeded in are based on the number of tapes used. All the machines ran on aa_, aabb_, bb_, and bbaa_ independent string inputs. As you can see on the output table, the flip machine does what it is supposed to do (a would be flipped to b and b would be flipped to a). The right shift machine also does what it is supposed to do (any input string would be shifted one place to the right and a $ is tacked on the very left of the input string). The flip and right machine just does the functionality of both flip and right shift machines. All test simulations ended up being accepted and the number of transitions are shown to the right of it.

Program 3: Other… 2 Stack Turing Machine
The output table is just the required information. It's notable that Number of configurations explored are the same, making average nondeterminism 1. When the main script is run you can see the code stepping through the manipulation of the two stack TM with the stacks and how it pushes and pops the symbols. Additionally when the test script is run the unittest outputs can be seen.

14 How team was organized

Each team member developed one program independently, allowing each of us to manage our own sub-environment. This made the collaboration process much simpler, but it also made it more difficult to help each other, because our code

	bases were completely independent.
15	What you might do differently if you did the project again **Something that might have made this project a bit easier would be developing some standardized way of representing and loading in different types of turing machines in a synced way. Developing a standard system would allow us to work together on a more flexible turing machine loader / simulator, that could be used across the sub-projects, instead of each developing our own solutions.**
16	Any additional material: **None**