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"

<table>
<tr><td>1</td><td>Team Name: chealy5</td></tr>
<tr><td>2</td><td>Team members names and netids: Catherine Healy
NetID: chealy5</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects:
Tracing NTM Behavior</td></tr>
<tr><td>4</td><td>Overall success of the project:
Overall, the project was successful and the code successfully operated and produced the desired files.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 15</td></tr>
<tr><td>6</td><td>Link to github repository:
https://github.com/CatherineH3/Theory_of_Computing_Project02/tree/main</td></tr>
<tr><td>7</td><td>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.
Note; I am really confused about how to make folders in git, so I will just make my best guess

<table>
<tr><td>File/folder Name</td><td>File Contents and Use</td></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td>traceTM_chealy5.py</td><td>Main NTM code</td></tr>
<tr><td colspan="2">Test Files</td></tr>
<tr><td>check_01_chealy5.csv
check_02_chealy5.csv
check_03_chealy5.csv
check_04_.chealy5csv</td><td>These are the test files. They include information on the Turing machine for different machines. Some were provided by instructor that were made by other students.</td></tr>
<tr><td colspan="2">Output Files</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td>output_01_chealy5.csv
output_02_chealy5.csv
output_03_chealy5.csv
output_04_chealy5.csv</td><td>This output file includes the output of the program including the trace for each machine and if it was rejected or accepted.</td></tr>
<tr><td colspan="2">Plots (as needed) / table</td></tr>
<tr><td>table_01_chealy5.csv
table_02_chealy5.csv
table_03_chealy5.csv
table_04_chealy5.csv</td><td>This table contains all the results for each input string and whether it was accepted/rejected, average nondeterminism, depth, and a number of configurations.</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries:
Programming languages: python
Libraries: The libraries I used were dataclass library to define some special data instructions and the deque library to implement the queue that helped me do breadth-first search. The csv library is also used to read and write to csvs.</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):

I created a Configuration dataclass to track all the possible configurations that could occur. The attributes were the tape contents, current state, and the characters remaining to process.

Additionally, I created a Turing Machine dataclass. It held all the information about the machine obtained from the input file, including the machine name, the start and accept states, and the transitions.

All of the transitions were stored in a 2-D transitions dictionary. It was structured follows: transitions[current_state][current_char] = (next_state, write_char, direction). This stored all of the transitions for the Turing Machine. It allowed for easy searching of the transitions with the indexing. The dictionary had to be 2D because there are multiple transitions for each state when it is nondeterministic, so there are 2 “keys” that must be searched to get the value, which was a 3-Tuple.

Lastly, an important data structure I used was the queue. This was used to do the breadth-first search algorithm and allowed me to push and pop in the queue and keep the items in order. This ensured that I was going level by level on the tree and making sure I hit the entire level before going deeper into the tree.</td></tr>
<tr><td>10</td><td>General operation of code (for each subproject)</td></tr>
</table>

	The user is prompted to enter a file name for the Turing Machine information file, along with a list of input strings. Additionally, the user may opt to enter the -t flag followed by an int to indicate the maximum depth. Then, the file is used to create the Turing Machine data class. This involves a lot of parsing and checking to make sure all of the information is being stored in the correct variable. Notably, I also use a dictionary with 2 keys to store all of the transitions. The key is the current state and current char, and the value is a 3 tuple with the next_state, the char to write, and the L/R direction. `transitions[current_state][current_char]=(next_state, write_char, direction)` After all the inputs are collected and formatted, the program loops through each input string. For each string, the read_input function is called. In this function, the initial configuration is set and placed into the queue to begin the Breadth-First Search. Next, a while loop is entered with the condition that it will continue unless the queue is empty or the max depth is reached. To achieve the breadth-first search, every configuration at the current depth stored in the queue is processed before moving to a deeper depth. For each configuration, the following information is popped from the queue: current_config, current_path, current_transitions = queue.popleft() The transition is searched for in the transition dictionary, and if found, the new configuration can be figured out. Also, the tape is updated based on the instruction (going R/L and writing characters) If the transition is not found in the transition dictionary, it is assumed that it goes to the reject state Also, throughout the process, the entire path is stored in a list of lists. If at any point it reaches the accept state or all the possible paths are rejected, it stops trying to generate more configurations and prints the results. All of the results are outputted with the statistics and information about depth, number of configurations, etc. A lot of effort was put into proper formatting for the results. Some are put in an output file, while the other results are printed into a table to meet the project specifications.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. ***Note about correctness:*** *Some of my depth/configurations/transition values may be off from what is expected by the grader. This is because I set up some of my code in a unique way so that what is considered the "root" of my tree and other results may be a little different from expected because I may have added a few extra configurations at the beginning and end. I also calculated the nondeterminism in a specific way. I attempted to add some explanations in various places in the documentation explaining potential discrepancies and why it actually makes sense based on the way I set up my code. It was not super clear in the instructions about what transitions should be counted as the start, etc. Within the realm of the definitions I used, the results should make sense.* Test case 1: DTM a+

	Why I used it: I wanted to do a DTM and compare it with a NTM to see how they compare What it told me about correctness: The simulator appropriately accepted and rejected the strings that it was supposed to. In the output file, I can see that the total # of transitions = the tree depth, which is expected for a deterministic TM. Also, the Nondeterminism measured on the accepted strings was 1, which is correct for something that is deterministic. The depth of the tree is based on the transitions to go deeper. Given that each path starts with the configuration of being empty and adding the next character, an input string of 1 would have a value of 1 because it takes one transition to go from that state of reading nothing to processing the one character. For the non-deterministic, the configurations explored should not be more than 1 of the depth (note: configurations in my code are like the nodes, and the rest is like transitions in the tree). So it makes sense that all the configurations are bigger than depth by 1 in the deterministic case. It should not be greater than that because there should only be one configuration per level because it is deterministic and should only be able to travel to the next state. It also makes sense that the rejected states have a slightly lower nondeterminism score because if they reject, they do not continue exploring for any other configurations and stop that path. Test case 2: NDTM a+ Why I used it: First attempt at testing a Nondeterministic case. What it told me about correctness: The simulator appropriately accepted and rejected the strings that it was supposed to. I was able to see that it traced multiple paths because it gave the longest path even in the rejected case. The calculated nondeterminism score was greater than 1 for all I also used thas to verify that it would stop if it looped too long. I set the max depth to be lower and typed in a very long string. The program correctly stopped the execution and identified that the maximum depth was reached. This is viewed in the table in the “Ran too long” part. Test case 3: NTM a*b*c* Why I used it: I wanted to try another NTM than the earlier to ensure that my project worked for other NTMs other than the first one I had tested on. What it told me about correctness: The simulator appropriately accepted and rejected the strings that it was supposed to. It also correctly identified a much higher average nondeterminism. Test case 4: DTM a*b*c* Why I used it: I wanted to compare the previous machine in case 3 with the deterministic version. What it told me about correctness: The simulator appropriately accepted and rejected the strings that it was supposed to. Also, for all the accepted strings, the calculated nondeterminism was 1, which is correct. Also, the depth equaled the total number of transitions. Note: For my nondeterminism calculation, it was a little confusing because Kogge had

	gave different formulas that conflicted. I eventually used the formula in the instructions, which said average nondeterminism =average(# configurations per level/ tree depth). According to the instructions, the goal of this number is to show the average number of new configurations that come from the average configuration and that a “1” is a completely deterministic solution. Note: The tree depth and number of configurations may appear different than expected, but this is how I intentionally set up my code. By convention, tree depth starts at 0 for just a root node and is equal to the number of transitions in the longest path. Because I started counting tree depth from 0 and not 1 and in order for my solution to make sense and align with this definition, I added 1 to the formula. However, my tree depth may appear greater than what the user may initially expect, because I include the configurations of before and after processing in the tree. This adds an additional transition than what may be expected, but this is intentional and can be understood better by viewing the output file.
12	How you managed the code development: I initially tried to code it on my own but got extremely confused. I went to office hours and talked to TA Tram Trihn about ideas for the code. She helped me understand the overall big picture of the code. She told me that that using data classes and certain data structures would help simplify things greatly. After thinking about and defining my data classes, I worked on the reading of the input file to the data structures. I then focused on making functions that could help process the tape information and track the various configurations. I also talked to Professor Kogge to better understand breadth-first search and how to use a queue and was able to implement that. After completing the majority of the code, I ran a bunch of test cases and had to go back and fix a lot of edge cases and errors I missed. I spent a long time running test cases and trying to debug.
13	Detailed discussion of results: *Determinisitc Turing Machine vs. Nondeterministic Turing Machines:* I ran 5 tests machines on my program, each with a variety of strings. Test #1 was a DTM for a+, which Test #2 was a NTM for a+. By having two different machines for the same language, I thought this would allow for a good comparison of the results. The first program I tested was the a+ Turing machine. I tested on both a deterministic and a nondeterministic version. table_01_chealy5.csv (a+ Deterministic machine):

TM Name	Input String	Result	Depth of Tree	Configurations Explored	Nondeterminism
a plus DTM	aaaaa	Accepted	5	6	1.0
a plus DTM	a	Accepted	1	2	1.0
a plus DTM	ab	Rejected	1	2	0.67
a plus DTM	aaaaa	Accepted	5	6	1.0
a plus DTM	aaabb	Rejected	3	4	0.8

Table_02_chealy5.csv (a+ Nondeterministic)

	Machine Name	Input String	Result	Depth of Tree	Configurations Explored	Nondeterminism
2	a plus Nondeterministic	aaaaaaaa	Accepted	8	37	1.44
3	a plus Nondeterministic	a	Accepted	1	2	1.0
4	a plus Nondeterministic	ab	Rejected	1	3	1.33
5	a plus Nondeterministic	aaa	Accepted	3	7	1.42
6	a plus Nondeterministic	aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	Ran Too Long	19	210	1.52

We can see that the program correctly accepts all strings composed of only a's and rejects strings with anything other than only a's. Also, I altered the max allowable tree depth using the -t flag to test that it would stop if it ran for too long for input string Aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa. It correctly said "Ran Too Long" instead of merely going to an accept or reject state.

I calculated the non-determinism by finding the average of the # of configurations per level / depth. Note that the nondeterminism score for all of the accepted strings in table_01_chelay5.csv is all 1. This makes sense, because each configuration can produce only one more configuration if it is deterministic because each state has only one possible transition per unique input. A much higher level of nondeterminism can be observed in the nondeterministic case. All values that are accepted are greater than 1 for the nondeterminism calculation, which is to be expected because each transition will produce more transition

Note: For rejected strings, it may seem odd that the nondeterminism is less than 1, but it is that way based on how I set up my code. This is because I defined nondeterminisim as the average number of new configurations that come from an average configuration based on the directions. If it is rejected, that means that no 0 new configurations were produced, so this would bring the number of new configurations down lower below 1 for these specific cases. I was unsure if this was how we were supposed to set up the calculation because the instructions were somewhat unclear, so I just ended up doing it this way.

Also, the tree depth may appear to seem off, but it makes sense based on the way I defined the tree. Note that in the way I defined it in my code, a single root starts at depth 0. Depth is added as transitions that go beyond the currently level are added. However,

an input with a single character, such as ‘a’, does end up having an depth of 1. This is because with the way I set up my code, there is an extra configuration of the tape before it even starts processing and after it is finished.

One pattern to observe that does make sense is that for the deterministic case, the number of configurations of the tree is 1 more than the depth for all the scenarios. This makes sense, because the depth is based on the transitions to another level, while the number of configurations is like the nodes of the tree. In the nondeterministic case, there will always be 1 more configuration than the depth. Also, to confirm correctness of my deterministic case, I also observed in teh output file that the depth always is equal to the number of transitions.

I used two additional test cases to confirm that the results above were consistent and could be reproduced with a different machine, a*b*c*. The machine correctly accepted and rejected the appropriate strings. As you can see, similar to the above, the accepted deterministic strings all have an average nondeterminism score of 1 (the exception is the rejected string, because it produces 0 configurations, contributing to a lower score based on the way I set up the code).
The Nondeterminisitc cases all have significantly higher scores, which is to be expected. Also, for the deterministic case, there is always 1 more configuration explored than the depth. (Note that my Nondeterminism calculation uses AVERAGE configuration, which is a separate calculation for each level , so “Configruations explored” in my table is a different metric.

Table_03_chealy5.csv (NONDETERMINISTIC)

1	Machine Name	Input String	Result	Depth of Tree	Configurations Explored	Nondeterminism
2	a*b*c* Nondeterministic	bc	Accepted	2	6	2.0
3	a*b*c* Nondeterministic	a	Accepted	1	2	1.0
4	a*b*c* Nondeterministic	abc	Accepted	3	15	3.12
5	a*b*c* Nondeterministic	aaaaabccccc	Accepted	11	173	3.28
6	a*b*c* Nondeterministic	aaaaaaaaaaaaaaaaaaaaaaaccccccccccccc	Accepted	37	1958	3.42
7	a*b*c* Nondeterministic	ca	Rejected	1	3	1.0

Table_04_chealy5.csv (DETERMINISTIC)

1	Machine Name	Input String	Result	Depth of Tree	Configurations Explored	Nondeterminism
2	a*b*c* Deterministic	bc	Accepted	2	3	1.0
3	a*b*c* Deterministic	a	Accepted	1	2	1.0
4	a*b*c* Deterministic	abc	Accepted	3	4	1.0
5	a*b*c* Deterministic	aaaaaaaaaaaaaaaaaaaaaaaccccccccccccc	Accepted	37	38	1.0
6	a*b*c* Deterministic	ca	Rejected	1	2	0.67

14 How team was organized:

I as an individual team, so the organization was pretty simple because I did everything myself. I first attempted to code the project, but had trouble and went to office hours.

	After getting some advice from the TA, I restarted the coding. I first focused on defining all of the data structures and reading the data to store the transitions and create the Turing machine. I then focused on creating helper functions to process the tape and store the configurations. I also wrote the breadth-first search algorithm with the queue. I then tested the code with a variety of test cases and spent a long time debugging based on the test results. I went back and revised the code after the test cases. Finally, I wrote the entire evaluation.
15	What you might do differently if you did the project again: What may be improved is that I need to start my projects earlier in the future. I waited until only a few days before the assignment was due to start, and this caused a high level of stress. I also need to be better at planning my code structure in advance. Initially, I structured it poorly and wrote everything all in one giant function. I got so confused that I had to restart from scratch and went and talked to a TA during office hours and it worked better the second time with data classes and data structures to help organize the code.
16	Any additional material: **Note: I submitted the project a little late but emailed Prof. Kogge about it. Professor Kogge I could submit it late and write a note that I emailed him on the submission.**