Project 2 Readme Team noonan

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: noonan</td></tr>
<tr><td>2</td><td>Team members names and netids: N/A</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: Successful, was able to demonstrate the behavior of the NTMs with the different user input strings</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 9.5</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/snoonan2/project2_noonan</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.
<table>
<tr><td>File/folder Name</td><td>File Contents and Use</td></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td>code_traceTM_noonan.py</td><td>This is my main python code file that contains all of the classes and functions I have created to show the behavior of nondeterministic turing machines. This program is able to read in the csv files of the machines, begin a configuration of the machines starting state and tape contents, then computes the possible transitions made through the breadth-first search algorithm. This file holds the code that calculates the analytics of the turing machine that are used to display to the user and examples seen in the table below. We use this file to see the behavior of the csv files constructed based on NTMs.</td></tr>
<tr><td colspan="2">Test Files</td></tr>
<tr><td>data_aplus_NTM.csv
data_binarystring_NTM.csv
data_equal10s_NTM.csv</td><td>These csv files are the representation of the NTM machines that are traced through in the program when chosen at command line by the user. The autonoma file consists of: name of the machine, list of state names for Q, List of characters from Σ, List of characters from Γ, The start state, Accept state, Reject state, and list of transition lines.</td></tr>
</table>
</td></tr>
</table>

		The data_aplus_NTM csv file constructs the tuple and transitions for the autonoma that would accept strings in the language a+ and reject those that are not. **The data_binarystring_NTM.csv file constructs the tuple and transitions for the autonoma that would accept strings that there is a 1 in the binary string that can be switched to a 0. If there is a 1 in the binary string, it will accept, and reject if not.** **The data_equal10s_NTM.csv constructs the tuple and transition lines for the autonoma that accept the strings that have equal number of 1s and 0s with 1s and 0s being the only characters in the alphabet. If there is not an equal number of 1s and 0s the machine will reject.** **These three machines are non deterministic meaning that there are multiple transitions out of a state for the same current character on the tape. The non deterministic machine can also accept the epsilon as a transition. The tracing of the NTM behaviors seen in my program shows the non deterministic behavior of the three machine csv files.**
	Output Files	
	output_terminal.png	**The outputs can be seen in the terminal after each run of the program. A description of analytics of a couple runs of the program using different NTMs and strings is located in the table below. I have included a screenshot of the terminal of an example to show that the program explored all the possible alternatives properly.**
	Plots (as needed)	
	plot_table_noonan.xslx	**This file contains a table of the analytics asked by Professor Kogge of the input strings and NTMs used in each run of the program. The table has three different NTMs with three different input strings. The table has the columns: NTM used, String used, Result (Accepted/rejected, ran too long), depth of tree, Number of configurations explored, average nondeterminism, and comments on what the strings output shows for the overall purpose of displaying the NTM behavior.**
8	Programming languages used, and associated libraries: **The programming language I used for this project was Python.** **Libraries I used:** **csv - Used for reading the Turing machine description from CSV files** **- Handles parsing of the machine configuration and transition rules**	

	argparse - Handles command-line argument parsing **typing - specifically List, Tuple, Dict, Set that helps with documentation** **dataclasses - Used for creating data classes with the @dataclass decorator** **- Simplifies class definitions for Transition and Configuration in my program** **My program only relies on the Python standard library with no other packages required which made the construction simpler.**
9	Key data structures (for each sub-project): **1.Core Machine Data Structures:** **Transition (dataclass): represents a single transition rule, immutable dataclass for storing state transition information, and used to define how the machine moves from one state to another** **Configuration (dataclass): Represents a complete snapshot of the machine, tracks tape contents on both sides of head, and forms a linked list structure through parent reference for path tracking** **2.Turing Machine Class Data Structures:** **Machine Definition Storage: Uses sets for efficient membership testing of states and symbols, dictionary with tuple keys for fast transition lookup, and lists for multiple possible transitions** **3. Computation Tracking Structures:** **BFS Level Tracking: 2D list structure, outer list represents depths/levels, inner lists contain all configurations at that depth** **Metrics Storage: Simple numeric types for counting and measuring, and list for storing branching factors at each level** **4. Implementation Structures:** **Transition Key Structure: Tuple used as dictionary key for efficient lookup and combines state and input for unique identification** **Parent Reference: Creates a tree structure through Configuration objects, allows backtracking to reconstruct successful paths, where each configuration points to its parent**

10	General operation of code (for each subproject) **My program shows the behavior of a non-deterministic Turing machine that explores multiple possible computation paths simultaneously. The program starts by reading a machine description from a CSV file, which contains the machine's states, alphabets, and transition rules. When given an input string, it creates an initial configuration representing the machine's starting state and tape contents. For the computation I use a breadth-first search strategy to explore all possible paths. At each step, it generates new configurations based on the available transitions, keeping track of all paths through parent references. This continues until one of three outcomes: reaching an accepting state (success), going through all possible paths (rejection), or hitting the maximum depth limit (timeout).** **Throughout the program, the system maintains metrics like the number of configurations explored, computation depth, and branching factors for later comparison in my table. For successful computations, my code can reconstruct the exact path taken to reach acceptance using the parent references stored in each configuration.It is controlled through a command-line interface where a user specifies the machine description file (CSV) and input string. The output includes both the computation result and detailed statistics about the machine's behavior during processing**
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. **I had three different csv files that represented three different Non Deterministic Turing machines that could best be tested on with a variety of user input strings. They were csv files that represented the autonoma for a+, equal number of 1s and 0s, and the presence of a 1 in a binary string. I used these autonoma csv files because they constructed machines that could be traced consisting of non deterministic transition paths. I had to also convert the machines into the proper csv file format per the project instructions of defining the formal tuple of the machine and writing the transition lines underneath. This took time to ensure the accuracy of the tracing of the machines. Using the csv files of the NTMs showed the correctness of my code by properly outputting the traces of the strings inputted by the user. This includes outputting if the string would be accepted/rejected, the right depth explored, and the non deterministic value based on the machine. Having three different autonoma files to test on my program displays that my program can trace a variety of machines and is easily applicable to correctly formatted NTM csv files. It is critical to have these csv files to run the program, so the correctness of the autonoma is important to seeing the correctness of the tracing behavior program.**
12	How you managed the code development **For the development of my project, I approached it by first establishing the core data structures needed to represent the machine's components. By using Python's dataclasses, I created a clean foundation that made the subsequent logic easier to implement. The development process started by first implementing the basic machine representation and file loading, then adding the core**

	computation logic for handling transitions and configurations, and finally building the more complex features like non-deterministic path exploration and metrics tracking. The breadth-first search implementation was particularly challenging, requiring careful consideration of how to track multiple computation paths simultaneously while maintaining parent references for path reconstruction. It was based really on my basic knowledge of the BFS algorithm. Testing was integrated throughout the development process, using simple example machines with csv files and inputs to verify correct behavior before adding more complex features. The command-line interface was added last, once the core functionality was stable and well-tested. I added documentation throughout my process of coding my project to ensure I knew what functions were executing which part of the process.
13	Detailed discussion of results: **My project successfully implements a non-deterministic Turing machine that can handle multiple computation paths simultaneously. It processes input strings according to machine descriptions loaded from CSV files and can determine whether inputs are accepted or rejected. The ability to handle non-determinism means it can explore multiple possible transitions from any given state, making it more powerful than a deterministic Turing machine simulator. My code incorporates metrics and analysis capabilities.. For metrics, it tracks total configurations explored, maximum tree depth, and configurations per level, while also measuring non-determinism through branching factors and path reconstruction. It achieves efficient performance through split string tape representation, dictionary-based lookups, breadth-first search for finding shortest accepting paths, and parent reference tracking. These features provide valuable insights into machine behavior patterns, computational complexity, and resource requirements. The output provides detailed output analysis including computation results (accept/reject/timeout), step counts, complete paths for accepted inputs, and comprehensive statistics. The combination of theoretical correctness and practical usability makes it a valuable tool for understanding and analyzing NTMS computations and behavior.**
14	How team was organized **The team was organized as a solo student doing the whole project. I felt that this worked well for the first project so I decided to do another project alone. I was able to properly manage my time creating the CSV files as well as constructing the code in parallel. I was also able to send time debugging and going to office hours when I had the ability in my schedule. I feel that I started the project early before break to get a head start. Working by myself meant some solo debugging, but I also utilized going to office hours to ask advice from Professor Kogge.**
15	What you might do differently if you did the project again **If I were to do the project again I would first focus on the correctness of my CSV files construction with the transitions of the machine listed. I ran into issues running my program thinking the errors were in my main code but in reality I had to fix the CSV files. I would also have asked earlier if we needed to provide tables or graphs for our results for our project so I wasn't including or working on the table at the end of the project. As I stated before, although I thought that I worked well as a solo team, I think I would have gained experience of coding**

	collaboration in a team setting. I want to get better at using GitHub pretty fluidly which comes with group work experience.
16	Any additional material: