**Team name:** ncichosk

**Names of all team members:** Nicholas Cichoski

**Link to GitHub repository:** https://github.com/ncichosk/theory-2

**Which project options were attempted:** Tracing NTM Behavior

**Approximately total time spent on the project:** 12 hours

**The language you used, and a list of libraries you invoked:** python; sys, csv, prettytable

**How would a TA run your program (did you provide a script to run a test case?):**

In order to run my program a TA would call the main file and enter four command line arguments. The first is the CSV file containing the machine instructions, the second is the desired output file, and the third is the maximum depth the program should map in the tree. The desired output file can be a text file, or the user can type "none" to have output printed directly to stdout. The maximum depth is also an optional argument; if nothing is entered for this command line argument, the program will default to a maximum depth of 100. Overall, the command line input for someone in the main directory should look like this:

Code/main_ncichosk.py Data/data_a_plus_ncichosk.csv Output/output_a_plus_ncichosk.txt 50

This will run the a plus machine given in the assignment description with a maximum depth of 50 and output to a file called output_a_plus_ncichosk.csv

**A brief description of the key data structures you used, and how the program functioned:**

This program primarily uses arrays to organize the data. To start, the transitions for the machine are copied into a 2D array called "rules" from the CSV file. These rules along with information about starting, accept, and reject states are passed into the run_machine function along with an input string to test.

The run_machine function first initializes a 3D array called "tree" which it will eventually return; this is the tree that maps all of the states that the NTM goes through. Each item in this array is a 2D array of possible states that the machine could be in for that "layer" of the tree. This tree is initialized to be empty and then [["", start, input]] is appended to it. This is saying that for the first layer/head of the tree, the only node in it will be one at the start state with the head pointing at the start of the tape. From there, the program enters a while True loop which will break if all of the states are reject states, the maximum depth is achieved, or an accept flag indicates that the accept state has been found. In each cycle, the program takes each state/configuration in the previous layer of the tree and generates a 2D array of all of the possible states that could be

generated from it by calling the update function. These new configurations are then appended to a 2D array representing the next layer of the tree which is then appended to the 3D array representing the tree.

The update function updates a single configuration given a set of rules. This function works by looking through the array of transitions and identifying transitions that would be taken given the state and head of the configuration passed in. These transitions are appended to an array of rules to be executed. Each of these rules is then applied to the configuration using the function update_config and appended to a list of new configurations which is returned.

The update_config function applies the rules of the rule passed into a given configuration by updating the configuration state, changing the character on the head, popping from the appropriate string, and appending that character to the other.

Two additional functions in tree_copy_funcs_ncichosk.py are layer_copy and tuple_copy. These are necessitated by the fact that Python will automatically have variables point to the same underlying data location when they are passed into functions or defined as one another (ex. Var1 = var2). As a result, the str() function is used to copy each tuple into a new data structure when making tuple copies. This function is then used to make copies of layers.

Finally, the results of run_machine are outputted to the appropriate variables and printed/written accordingly.

**A discussion as to what test cases you added and why you decided to add them (what did they tell you about the correctness of your code). Where did the data come from? (course website, handcrafted, a data generator, other):**

The test cases that are used in this project are different machine cases and different string inputs for those machines. For the machine options, I developed the code using only the a plus machine provided in the project description. This allowed me to develop my code without fear of a bug in a machine. After my code was mostly done, I decided to create my own machines. I created an a*b*c* and equal 01s machine. This allowed me to test my code on a variety of machine inputs. I also create deterministic versions of every machine to test that it correctly processed these with an average non-determinism of 1.

After these machines were created I also tested them for different string inputs. I did this by methodically choosing string inputs that would test every transition of a given state machine. I also added random input and really long input to test halting cases. This process ensured that the machines I created and the project code could correctly process any transition necessary to decide a string.

**An analysis of the results, such as if timings were called for, which plots showed what? What was the approximate complexity of your program?**

For my result analysis, I used the prettytable library to output key metrics of each run (machine name, input string, decision, depth, configurations searched, average non-determinism, and comments). All tables in the Tables folder except for table_combined_ncichosk.txt are directly generated from code output. Table_combined_ncichosk.txt is a manually created table where I combined one run from each machine for comparison; a copy of this can be seen at the bottom of this readme.

What I found in my analysis is that each machine varied in its level of non-determinism. At the lowest level were the deterministic versions of each machine. These each had an average non-determinism of 1 (calculated by taking configurations explored / tree depth). This is expected as deterministic machines will only test a single configuration for each layer of the tree.

The tables for the non-deterministic machines reveal that input strings have differing levels of non-determinism with shorter strings tending to have less non-determinism and longer complex strings having more. Looking at the text output in the Output folder, we can see the configuration trees generated by each input string for each machine. We can see from these trees that each machine has a maximum level of non-determinism or a maximum number of configurations that the machine is in at once. For all deterministic machines this is 1, for the non-deterministic a plus machine this appears to be 3, for the non-deterministic equal 01s machine, this appears to be 2, and for the non-deterministic a*b*c* machine this appears to be 10.

These differences in “maximum non-determinism” demonstrate why the average non-determinism of each machine seems to differ. At the bottom of each machine's output table, there is an aggregated level of non-determinism. For the a plus machine, this tends to be just under 3 for a large sample size, for equal 01s it tends to be just under 2, and for a*b*c* it is usually between 6 and 7 for large samples. This shows that some machines are more non-deterministic than others with a*b*c* being the most and equal 01s being the least.

Overall, the output files in the Output and Table folder reveal varying levels of non-determinism between different machines and different input strings. Machines with more potential states to be in tend to have more non-determinism as well as longer input strings.

**A description of how you managed the code development and testing.**

I broke down the development of my code into several steps that I could develop and test before getting too deep into the code. For the development of my program, I used the a_plus.csv machine that was provided in the project description as I did not have to worry about the accuracy of the machine when testing.

The development step I took was to make sure I could read in the machine correctly. For this step, I developed lines 31-60 of the main function. I used print statements to make sure the array of rules and machine information that was being read in from the machine file was correct.

The next step that I took was to develop the run machine function. Because I did not have any update function, I simply returned a copy of the layer instead of updating the configurations. For this stage, I also set a five-layer limit on any execution. This allowed me to focus on developing the mechanism for moving from layer to layer.

After I was able to move between layers, I created the update and update_config functions. Again I had a five-layer limit on the machine runs. As I developed this mechanism, I noticed a natural split of updating a single configuration and updating a whole layer so I split it into two functions. I tested this function by running the machine and following the route through handwritten state diagrams and confirming the tape and states were correct. I also caught a few bugs in one of the machines I created through this process.

After the run_machine, update, and update_config functions were created, I noticed an error in how the program was copying configurations, this made it so the code outputted a tree with the same configuration at every node. To fix this, I developed the layer_copy and tuple_copy functions found in tree_copy_funcs_ncichosk.py

Lastly, I formatted my output to clearly display the key metrics of each run in both text output as well as a table. I then tested and debugged my code for a variety of machines and input strings.

**Did you do any extra programs, or attempt any extra test cases:**

Yes, I created five extra test programs, two of which are NTMs and three of which are DTMs that accompany each NTM program (including the provided a plus machine). I created these by first drawing out the Turing machines I wanted to create on paper. After that, I defined the first seven lines of the machine by taking note of state names and all of the characters used in input and written onto the tape. I then recorded each transition as a rule in the CSV file. Lastly, to test these new machines, I tested strings that would take every possible transition. For example in my a*b*c* machine, I tested the strings _, a, b, c, ab, bc, ac, abc, ba, ca, cb, and aabbcc. I then verified that these strings were accepted or rejected in an appropriate manner. After the testing was complete, I shared these machines to be used by other students. I did this process for all new machines I programmed.

**Table of sample input from table_combines_ncichosk.txt:**

| Machine | Input String | Result | Depth | Configurations Explored | Average Non-Determinism |
|---|---|---|---|---|---|
| a plus DTM | aaaaa | Accepted | 6 | 6 | 1 |
| a plus | aaaaa | Accepted | 6 | 16 | 2.6666 |
| abc star DTM | aabbcc | Accepted | 7 | 7 | 1 |
| abc star | aabbcc | Accepted | 7 | 47 | 6.7142 |
| equal 01s DTM | 110110 | Rejected | 21 | 21 | 1 |
| equal 01s | 110100 | Accepted | 29 | 42 | 1.4482 |