Readme_aarnett2

Version 1 8/22/24

A copy of this file should be included in the github repository with your project. Change the teamname above to your own

1. Team name:
   Aarnett2

2. Names of all team members:
   Anna Arnett

3. Link to github repository:
   https://github.com/anna-arnett/NTM_Tracing

4. Which project options were attempted:
   Program 1: Tracing NTM Behavior

5. Approximately total time spent on project: **~12**
6.
7. The language you used, and a list of libraries you invoked. **Language used: Python. Libraries invoked: csv, collections**

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

   A TA would run my program by running one of the check…aarnett2.py scripts. There is a test script for each NTM example that can be run on its own.

9. A brief description of the key data structures you used, and how the program functioned.

I used a dictionary (defaultdict) to store the NTM's transition table. The keys were tuples of the form (state, input_symbol). The values were lists of possible transitions, where each transition is represented by a tuple (next_state, write_symbol, direction). This made it easy to handle nondeterminism by storing multiple possible transitions for a single state and input symbol.

I also used a queue to implement the breadth-first search for exploring the NTM's configuration tree. It stores configurations of the form (left_of_head, current_state, right_of_head), which represents the tape contents and the machine's current state.

I also used a list of lists to store the configuration tree, where each level corresponds to a set of configurations after a given number of transitions.

The program functioned by reading in the NTM definition from a .csv file, and parsing the states, alphabet, start state, accept state, and transition rules into a defaultdict. The input string is then placed on the tape and the machine starts at the

initial configuration (“”, start_state, input_string). The queue is initialized with the starting configuration, and for each configuration in the current level, possible transitions are received from the transition table (defaultdict). For each transition, a new configuration is created by updating the tape, state, and head position. The new configuration is added to the next level of the tree. The program continues until either an accept state is reached, all paths lead to rejection, or the maximum step limit is reached.

10. 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 first test case I used was the a_plus.csv sample NTM that was given from the project instructions. I decided to add this test case because I knew it would be correct (since it was provided for the project) and there was also some guidance with it. So, I thought it would be a good starting point to test my code with. This test case allowed me to initially see that my code was accepting the correct input strings as well as rejecting the input strings it was supposed to reject. This was a good baseline with which to build my code off of. I also found that the measure of nondeterminism was consistent with the machine’s design. Also, for the rejection case, the simulator halts relatively quickly, so that demonstrates the efficiency of handling invalid inputs.

A second test case I used was the abc_star test case which processes strings that consist of zero or more a’s followed by zero or more b’s and then zero or more c’s. I added this test case to test nondeterminism handling, as this NTM is highly nondeterministic, with multiple transitions possible from a single state. This test case told me that my code correctly simulates multiple nondeterministic paths, and the measure of nondeterminism matched my expectations based on the number of transitions from each state. I also learned that my program was scalable and handled the increased depth from longer strings well. This data came from the shared google drive with test cases.

I also used the abc_equal test case, which recognizes strings that have an equal number of a’s, b’s, and c’s. This language was a little more restrictive than abc_star, so it allowed me to test how my code handles this type of pattern matching. This test case allowed me to see that my code was able to correctly handle strings of this nature. This data came from a data generator that allowed me to ensure the NTM was accurate so I could more accurately test my code.

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

My program correctly handles the acceptance and rejection of input strings based on the NTM’s transition rules. In the example of a_plus, for inputs like “aaa” and “aaaaa”, the machine properly explores all the nondeterministic paths and identifies the correct

accepting state. For an invalid input like "bbbbb", the program efficiently detects rejection after just a few steps, because no valid transitions exist for the input symbols. The depth of the configuration tree also matches the number of steps required to explore all paths. The measure of nondeterminism also seems consistent with the design of the NTM. As for complexity, the program simulates an NTM using breadth-first search, exploring all possible paths level by level. The time and space complexity depend on the length of the input string, the number of possible transitions at each configuration, and the depth of the configuration tree. The time complexity grows exponentially with d steps, resulting in b^d configurations. So, the time and space complexity are $O(b^d)$.

Results Description

Machine Name	Input string	Result	Depth	# configs explored	Avg non-determ.	Steps taken to reject
a_plus	aaa	accepted	5	8	2.00	-
a_plus	aaaaa	accepted	7	12	2.00	-
a_plus	bbbbb	rejected	-	-	1.00	2
abc_star	abc	accepted	5	17	2.43	-
abc_star	aaaabbbbccc	accepted	14	59	2.68	-
abc_star	aabcbcababca	rejected	-	-	3.17	6
abc_star	cccbbbaaa	rejected	-	-	2.33	5
equal_abcs	abc	accepted	5	5	1.25	-
equal_abcs	abcabcabc	accepted	11	11	1.10	-
equal_abcs	aaabbbcc	rejected	-	-	2.00	3
equal_abcs	aaaaabbbbbbcccc	rejected	-	-	2.00	3

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

I developed the code on my personal laptop in VSCode. I started with writing and testing the code based on the a_plus csv file, which as I described earlier was helpful because of the additional information on it given in the project description. A lot of the debugging was centered around the different data structures I was using and deciding how exactly I should represent the data in ways that could be read in from the .csv file. I used a lot of print statements in order to clearly display what was happening inside the program, as it was easy to get lost in minor details. Once I started to add more tests cases and confirm consistent functionality, I could work out smaller bugs. As I added more test cases, I added the files of these to my github repository to keep track of everything.

13. Did you do any extra programs, or attempted any extra test cases

Yes, in addition to the initial a_plus test case, I also tested with several other NTMs (including abc_star and check_equal_abcs). These additional tests were successful in helping me confirm the functionality of my code.