Regex Theory: Deterministic Finite Automata

Eckel, TJHSST AI2, Spring 2023

Background & Explanation

Regular expressions are practically used as a means of efficiently searching text for patterns, but their origin is from an entirely different corner of the computer science world. Regular expressions arose from theoretical computer science – a mathematical study of the theoretical capabilities of simple computers. You may have heard of a Turing machine, for instance; a Turing machine is not a physical computer, but a theoretical idea that can be mathematically proven to be equivalent to much more complex real computers. In this way, it's good for mathematical modeling.

We're going to explore one small corner of theoretical computer science and show how the concepts of regular expressions arose. Specifically, we're going to study **finite automata**. A **deterministic finite automaton** (DFA) is a theoretical computer that has a finite number of particular states. As it reads an input string one character at a time, each character causes it to transition from the current state to another according to preset rules. That is, for each state, for each possible character in the language it is designed to process, the machine takes one specific action. (ie, state 3, when it receives an "a", moves to state 7 – that sort of thing.) Certain states are defined as **final states**; if, at the end of processing the input string, the DFA is on a final state, then the input string is said to **match**.

First, you'll write code to emulate the execution of a DFA when given a specification in an input file. Then, you'll write your own DFAs to particular specifications.

You may not import `re` on this assignment. You'll see why on **Regex Theory: Write a (Simple) Regex Compiler**.

Input Specification

Your code will need to read in a txt file that contains the spec for a DFA. A DFA spec file will look like this:

```
ab
3
1 2

0
b 1

1
a 1
b 2

2
```

The first line contains the language – every character that may appear in an input string for this DFA. This may be arbitrarily long, but will not contain any repeated characters. (You can also assume no weird characters that might cause strange effects, like \n or \t.)

The next line contains the number of states in the DFA. Assume they're numbered starting from 0, and 0 is always the starting state. (So, since this file has a 3, there are three states – 0, 1, and 2.)

The next line contains, separated by spaces, all states that are final states.

After that, separated by blank lines, are the specifications of each state. After a line containing the number of the state is some number (possibly zero) lines that have a character from the language then a space then the state to transition to when given that character. You can assume that if a character in the alphabet does not appear in a state's definition, receiving that character on that state should lead to a non-match (ie, you may stop processing input and return False.)

To implement this in a flexible way, I recommend storing the state transitions in a dictionary of dictionaries. You can find a way to include information about which states are final in that dict or in another data structure; I chose a separate one. So, if I were hardcoding the same DFA specified in the file above, I would write it like this in Python:

```
dfa = {
    0: {
        "b": 1
    },
    1: {
        "a": 1,
        "b": 2
    },
    2: {}
}

final = [1, 2]
```

Your IDE should be able to format nested dictionaries like you see above. Write `dfa = {}` then place your cursor in between the two brackets and hit enter to see it automatically format / indent.

You can choose to store your DFAs differently, but the remaining assignments in this strand will assume you're using the form given above for the sake of simplicity.

Pro tip: to make parsing the input file a little easier, use the `.read()` command to read in the entire file as one big string, and split it on `"\n\n"`. This will separate each state into a separate string that can itself be split & parsed appropriately! Again: **you may not import `re` on this assignment**; you'll see why later on. You have to do this using Python's built in text manipulation functions.

Displaying a DFA to the Console

Nested dictionaries are notoriously difficult to print in any readable way. What I'd like you to do instead is write code to take however you're storing your DFAs' state transition information internally and output it as a table. Each ROW of the table should be a state in the DFA. Each COLUMN should be a letter in the language. If a state has a spec for what to do with each input letter, display it, otherwise put a _. For instance, the DFA above would be nicely printed as:

```
*       a       b
0               1
1       1       2
2               _
```

For the purposes of this assignment, you can assume that there will be less than 10 states in any DFA in the assignment which should make formatting this table pretty straightforward (since each state will end up specified by a single character).

Part 1: DFA Emulator

Once you have wrapped your head around the input spec & how to store and display a DFA's state transition information, your goal is to write code that does as follows:

1) Accept two command line inputs. The first is a .txt file containing the DFA specification. The second is a .txt file containing strings, one on each line, in the DFA's given alphabet.
2) Read in the DFA specification and store it (perhaps as shown above).
3) For each string in the second text file, run it through the DFA. That is, starting at state 0 in the DFA specification, read in one character from the input string at a time. Look up each character in the current state's dictionary to see where to move next. If the current character does not appear in the current state's dictionary, return False. Once the input string is processed, check to see if the current state is a final state. If so, return True. If not, return False.
4) Then, output:
    a. The DFA's state transition information, printed nicely in a table
    b. The list of final states in the DFA
    c. For each input string, the "True" or "False" value followed by the input string itself

Sample Run for Part 1

On the course website you'll find a txt file that contains the spec shown above, as well as a txt file of test cases. When you run your code on those two txt files, this is what it should output:

```
*           a           b
0           _           1
1           1           2
2           _           _
Final nodes: [1, 2]
False a
True  b
False aa
True  bb
False aba
True  bab
False abba
True  baab
False ababa
False babab
False abbbbb
True  baaaaa
False abbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbba
True  baaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
False abbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaa
False baaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabb
```

Part 2: Writing DFAs

Now, a challenge – below are seven descriptions of DFAs, and your job is to hardcode a DFA in your Python script that will follow each specification. I should see, in your .py file, clearly labelled, where each DFA is written.

Your goal for this section is to receive two command line arguments again, but this time the first argument is an integer specifying a problem number below. You'll use the DFA you've hardcoded as an answer to that question. The second command line argument will be a file of test cases, as before.

Your challenges are:

1) On the language "ab", create a DFA that matches on precisely and only the input string "aab".
2) On the language "012", create a DFA that matches any input string that ends in a "1". (Careful here; note that states will be specified as **integers** and input characters should be **characters**, so an input of "1" may take you to state 1, but those 1s are different data types.)
3) On the language "abc", create a DFA that matches any input string with a "b" in it somewhere.
4) On the language "01", create a DFA that matches any input string with an **even** number of "0"s (including none at all) and **any** number of "1"s (including none at all).
5) On the language "01", create a DFA that matches any input string with an **even** number of "0"s and an **even** number of "1"s.
6) On the language "abc", create a DFA that matches any input string that does not contain the particular substring "abc". (This should match on an empty string.)
7) On the language "01", create a DFA that matches on any input string that at some point contains the substring "10" and then, at some point **after** that, contains the substring "11".

You are not provided test cases for this, but it should not be tremendously difficult to create your own.

If you receive this set of command line arguments:

```
your_script.py 5 tests.txt
```

…that should cause your code to run your answer to number 5 on every string in the file tests.txt. Your output should be the same as part 1 – the state transition table, list of final nodes, and each test case with the True or False result prepended.

Specification

Submit a single Python script to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- **READ THIS CAREFULLY. Students messed this up frequently last year! This submission COMBINES both parts of this assignment.** Specifically, your code takes in two command line arguments and behaves as follows:
    - If the first argument can be converted to an integer (`try` / `except` might be helpful here) then run your code for part 2. Use the DFA specified by your answer to that integer's corresponding problem and run it on test cases given in the file in the second command line argument.
    - If the first argument cannot be converted to an integer then run your code for part 1. Read in the DFA specified in the file given in the first argument and run it on the test cases in the file given in the second argument.
    - In either case, nicely print everything required above – the state transition table, final list, and each test case with prepended True/False result.
- **You do not import `re`.**