COMP 370, Fall, 2020
Program #2, NFA to DFA conversion
Date Assigned: Wednesday, September 9
Date Due: Thursday, October 1, 10PM
Possible Points: 100

For this assignment you will write a Python 3 program (in a file named `pa2.py`) that defines a class named `NFA`. The constructor for this class reads an NFA specification from a file. The class also has a method that converts the NFA to an equivalent DFA, and writes the DFA to a file, using the same DFA file format as specified in pa1. Your program should behave exactly as described further below.

1 Initial Setup

Both you and your partner will need to clone the starter code for your group using git. The following assumes that you are using VSCode as your IDE. If you are using some other IDE, follow the instructions for that IDE to clone a repository, or just use command line git ("git clone url").

1. You will need your group number in what follows. Get this from the PSA Group Numbers file under the Programming Assignments tab on Blackboard. This file also has your assigned partner.

2. In VS Code, open the command palette and select the Git Clone option.

3. When prompted for the repository URL, enter the following, with X replaced by your group number (e.g. 7 or 12):
   ssh://git@code.sandiego.edu/comp370-fa20-pa2-groupX

4. Choose the Open Repository option in the window that pops up in the lower-right corner of the screen. The repository should contain the following files:

   - `pa2.py`: You will write your code in this file. It includes the header for the `NFA` class that you will write, and the headers for the constructor and `simulate` methods that you must add.
   - `test_pa2.py`: This program will test your `pa2.py` program. See the discussion below on this.
   - `nfai.txt` for i = 1, ..., : Test nfa files.
   - `stri.txt` for i = 1, ... : Test input strings for the corresponding nfa's.
   - `correcti.txt` for i = 1, ... : Correct answer for the test input strings for the corresponding nfa's

5. Each programming team will have it own repository that has been initialized with the same starter code, so when you sync your code, you are sharing with your partners, but with no one else in the class. (I can see your repository also.)

6. Once you have cloned your repository, you should copy your (correct) pa1 code (`pa1.py`) into your repository, and add it to the repository. (See the instructions for pa0 for a reminder of how to add a new file to a repostiory.) The program `test_pa2.py` imports `pa1.py` to test your `pa2.py` code.

7. Remember that if you close VSCode, then when you reopen it, you should see your repository. But if you dont see it, then just select `File->Open`, and then select the directory containing your repository.

8. I also recommend that you stage changes, commit those changes, and sync the changes periodically as you are working on the program, and certainly when you are done with a session with your programming partner. This ensures that you wont lose any of your work in case your computer gets lost or a file gets accidentally deleted.

9. See the section below on my recommended approach for solving this problem. Your final code will not have to be too long, but it does require careful thought.

2 Program Specifications

As said above, your program should be contained in a file named `pa2.py`. In this file you should define a class called `NFA` (all capital letters). `NFA` should have a constructor that takes as its parameter the name of a file, and initializes the `NFA` from the contents of the file. (Specifications for the file format are further down.) `NFA` should also have a method named `toDFA` that takes as its parameter the name of a file, and writes to that file the specification of a DFA that is equivalent to the NFA that calls the method (the "self" DFA). The format of this file should be exactly as the DFA file format from the first programming assignment (pa1). The project repository contains a `pa2.py` file that has the headers for these.

The format of the NFA description is as follows (this is close to the same as the DFA specification in the first programming assignment, except that there can be nondeterminism, epsilon transitions, and a transition does not have to be specified for all state/symbol combinations):

1. An integer that is the number of states in the NFA. This appears by itself on the first line of input. (In the remainder of the description, each state must be referred to by an integer in the range $[1, 2, ..., n]$, where n is the number of states.)

2. The alphabet of the NFA. This appears by itself on the second line of input. Every character in the line (not including the terminating newline character) is a symbol of the alphabet. The alphabet cannot include the letter e, as this is reserved for specifying epsilon transitions.

3. The transition function of the NFA. There will be one line of input per possible transition in the transition function, starting with the third line of input. The format of an entry is

   ```
   qa 'c' qb
   ```

   This entry indicates that if the NFA is in state `qb` and the next symbol scanned is a `c`, then the NFA can transition to `qb`. `c` can also be the letter e, in which case is represents an epsilon transition.

   `qa` and `qb` must be valid states; that is, qa $\in \{1, 2, \ldots, n\}$ and qb $\in \{1, 2, \ldots, n\}$. `c` must be in the alphabet or be the letter e (representing epsilon), and the single quotes must be present (even for the letter e).

   The entries of the transition function can appear in any order.

4. A blank line terminates the transition function entries. We need this because we don't know in advance how many transition function entries there will be.

5. An integer that is the start state of the NFA. This appears by itself on the first line after the transition function lines.

6. The set of accept states of the NFA. These appear together on the line following the line containing the start state.

Entries on a line are separated by one or more space characters. The first entry on a line is preceded by 0 or more space characters, and the last entry on a line is followed by 0 or more space characters (in addition to the newline character).

Your program can assume that the input file will be correctly formatted, with no inconsistent data. (For example, if there are only 6 states in the NFA, there will be no transition function entries that specify a state of greater than 6 or less than 1.)

3 Working on the Program

You should work with your programming partner, using the pair programming software development technique.

As said above, the project repository contains test data for you. For each test NFA there is the NFA specification file (e.g. `nfa1.txt`), a file containing test strings (e.g. `str1.txt`), one string per line, and a file containing the correct answers for each test string (e.g., `correct1.txt`), one answer (Accept or Reject) per line. (The lines of `correct1.txt` correspond to the lines of `str1.txt`.)

The project repository contains a Python program called `test_pa2.py` that tests your DFA class on all of the test DFAs in the files described above, and it reports the results. Your program must pass all tests before you submit your program for grading. If it fails any test, I'll return the program to you for further work. There is not partial credit concerning correctness of your program.

As I said above, you should have added your solution code to pa1 (`pa1.py`) to your repository, as `test_pa2.py` imports that file, and uses it to test your code. Note that when I test your code, I will use my version of `pa1.py` to test your program. So make sure that the format of the DFA files you create exactly follow the specifications of pa1.

Your program should follow the programming style guidelines contained in the programmingStyleGuidelines.txt document on `blackboard->Programming Assignments`. Failure to follow these guidelines will result in points off.

To submit your program for grading, be sure that you've pushed it the repository, and post a note on campuswire, category `PA2_submit`, giving just your group number. (Questions about pa2 should go to category PA2.)

4 Recommended Approach

I recommend you approach the problem by following the following steps:

1. First, understand the NFA to DFA conversion algorithm. Review the algorithm as described in Theorem 1.39 in the textbook, starting on page 55, and the example of the construction described in the Theorem, in Example 1.41, starting at the bottom of page 56. I also go through an example in my virtual lecture 10: equivalence of NFAs and DFAs. Note that your program needs to account for epsilon transitions in an NFA. The end of Theorem 1.39 describes how to handle these, and Example 1.41 also handles these. (Note: I suggest that in your code, you use the approach I describe in my virtual lecture - generate the states of the DFA on an "as-needed" basis, rather than generating all possible DFA students (corresponding to all possible subset of the NFA states), as is done in Example 1.41.

2. On paper, trace the algorithm for at least one sample NFA in the repository, making sure you have at least one example that has epsilon transitions. If you come to office hours for help on this project, I'm going to ask to see the examples you traced. (If you cannot trace the algorithm on paper, then it will be hard for you to code it up.)

3. Familiarize yourself (if you haven't already) with the debugger in your IDE, particularly the features that allow you to set breakpoints and examine the values of data (variable, instance variables, parameters) as the program is executing.

4. Start coding by writing the `NFA` class constructor, which should create an internal representation of the NFA it is reading from the file. You internal representation of an NFA will likely be similar to the internal representation of a DFA from your pa1 program. Once written, run your debugger on the program, reading in from a couple of the test NFA files. Examine the values of the variables, to make sure that they correctly represent the NFAs. If you don't have a correct internal representation of the NFA, you're not going to be able to write out a correct DFA.

5. Next, convert the internal representation of the NFA into an internal representation of an equivalent DFA. (Probably your internal representation of a DFA here will be very similar to your internal DFA representation from pa1.)

6. Again, use the debugger to validate (for at least a couple test NFAs) the internal representation of your DFAs, debugging your code as necessary.

7. Next write the DFAs (again, for at least a couple of the test NFAs) out to their respective files. Look at them (by hand) to make sure they are correct, and have the correct format.

8. Complete the testing on a couple sample NFAs by running your pa1 code on the DFA files you've created, and make sure your pa1 code reports the correct answer to all of the test strings (accept or reject). Remember that the test code is in the files `nfai.txt`, `stri.txt` and `correcti.txt`.

9. Complete your testing by running `test_pa2.py` and making sure it reports all tests are correct. No credit unless you get all tests correct.

5 Late Penalties

The turn-in time for your program will be the timestamp on the last sync (for a correct program) that you did to your repository, and not the time you post your submit note on campuswire. If you turn in your program after the deadline, the following late penalties apply:

- After Thursday, October 1, 10PM, but before Monday, October 5, 10PM, 10 point penalty
- After Monday, October 5, 10PM, but before Thursday, October 8, 10PM, 20 point penalty
- After Thursday, October 8, 10PM, but before Monday, October 12, 10PM, 30 point penalty
- After Monday, October 12, 10PM, but before Thursday, October 15, 10PM, 40 point penalty
- After Thursday, October 15, 10PM, but before Thursday, November 12, 10PM, 50 point penalty
- After Thursday, November 12, 10PM, no credit.