Deterministic PDA Documentation

Thanh Son Phung

Oct 2, 2020

https://github.com/xandus1/Deterministic-PDA-emulator

1 Overview

In this project, I build the emulator for a deterministic PDA along with a few sample codes. The program is coded in Python 3.8, but I don't use any packages or syntax exclusively to Python 3.8 so I expect that it would work on any Python 3 version if you decide to test run it. The format of my code is as follows:

- main.py: This file includes the very basic call from the PDA class. It takes in two files - one describing the PDA, the other containing the input - and feeds these two files to an instance of a PDA. In details, the main file first builds the PDA with the given description of a PDA. The user would confirm that the PDA is constructed correctly by typing in 'y' (meaning yes). Then the main file will call an instance of PDA to execute the given input.
- PDA_machine.py: This file contains only the definition of the PDA class. It builds the PDA with utility functions for the main file to call if needed.
- input_PDAx_y.txt and PDA_machine_z.txt: x, y and z are placeholders for actual numbers for testing purpose. I prepare 3 PDA descriptions in the PDA_machine_z.txt, z is numbered from 1 to 3. The input files are input_PDAx_y.txt, where x is from 1 to 3, denoting which machine it is supposed to be fed to, and y is from 1 to 4, since for each PDA description I use 4 inputs to test it.

2 PDA and input format

The format of an input file is as follows:

- Exactly one line of letter strings ending with '$' to denote the end of input string. The '$' helps with the pop '$' from the stack when there's no input left. An example of an acceptable input is "abababa$".

To encode your PDA, you must enumerate all your states (i.e., translate them to numbers) from 1 to n. Furthermore, each transition can only be either pushed or popped.
The format of a PDA description is as follows:

- First line: The number of states in the PDA (e.g., 4.)
- Second line: The input alphabet of the PDA separated by the symbol ^ (e.g., a^b^$.)
- Third line: The stack alphabet of the PDA separated by the symbol ^. Note that 'E' is used to denote empty symbol. (e.g., a^b^E^$.) Note that the initial stack already has the symbol '$'.
- Fourth line: start state (e.g., 1.)
- Fifth line: the list of accept states, separated by empty space symbol (e.g., 2 3)
- Sixth line and above: for each line here, you must list a transition as follows (current state, input symbol, top of stack symbol, next state, new top of stack symbol.) Consecutive elements in the tuple are separated by an empty space symbol (e.g., 1 a E 2 a to denote on state 1, input a, you push symbol a onto the stack and transition to state 2.)

3 Execution

To run the program, you need to make sure all relevant files are in the same directory. After navigating to the directory using the terminal, you can run the program by the following code:

```
python3 main.py PDA_machine_X.txt input_PDAX_Y.txt
```

Again, X and Y are placeholders for some numbers. When you hit "Enter" on the above line, the terminal will first print out all information about the PDA to make sure that it interprets your PDA encoding correctly. Then it will ask for your confirmation that the PDA is interpreted correctly, so you must either type 'y' for yes or 'n' for no. The terminal message should be somewhat of this form:

```
    Building PDA...

There are 3 states in this PDA.
The input alphabet for this PDA is: a b $
The stack alphabet for this PDA is: a b $ E
The start state is: 1
The accept states are: 3
The transition table is as follows:
State 1, on input a and top of stack E, goes to state 1 with a as the
new top of stack.
```

```
State 1, on input b and top of stack a, goes to state 2 with E as the
new top of stack.
State 2, on input b and top of stack a, goes to state 2 with E as the
new top of stack.
State 2, on input $ and top of stack $, goes to state 3 with E as the
new top of stack.
State 2, on input a and top of stack a, goes to state 1 with E as the
new top of stack.
State 1, on input $ and top of stack $, goes to state 2 with E as the
new top of stack.
Building PDA is complete.

Please verify that PDA is constructed correctly. Proceed?[Y/N]
```

After you enter 'y', the program will execute with your given input. For each transition, the terminal will print the details of the transition and the state of the stack. In this process of executing, proper messages are printed on whether the input string is accepted by the PDA or not and why. The executing message should look something like this:

```
    OK.
Executing PDA with given input: aaabbbba$

On state 1, input a, we push a on to the stack and goes to state 1
The stack now, from bottom to top, is:
$ a
On state 1, input a, we push a on to the stack and goes to state 1
The stack now, from bottom to top, is:
$ a a
On state 1, input a, we push a on to the stack and goes to state 1
The stack now, from bottom to top, is:
$ a a a
On state 1, input b, we pop a from the stack and goes to state 2
The stack now, from bottom to top, is:
$ a a
On state 2, input b, we pop a from the stack and goes to state 2
The stack now, from bottom to top, is:
$ a
On state 2, input b, we pop a from the stack and goes to state 2
The stack now, from bottom to top, is:
$
There exists a transition in the table for state 2 and symbol b, but top
of stack doesn't match the transition.
```

```
Thus, we reject.
```

The program is designed to catch some common typos, but it isn't resilient to malicious attempts or bad inputs. Thus I assume that the user is well-behaved. In the end, the program will simulate the PDA and decide whether an input is accepted by the PDA or not. Thus, we have (hopefully) achieved the goal of building a deterministic PDA emulator.

```
    ----- END OF DOCUMENTATION -----
```