CS 361: Project 1 Part 2 (Nondeterministic Finite Automata)

Due date: November 13th, 2018

Fall 2018

1 Project Overview

In this project you will implement two classes that model an instance of a nondeterministic finite automaton (NFA) including a method that computes an equivalent DFA (Algorithm in Theorem 1.39) to this NFA.

2 Objectives

- Continue to strengthen the concept of packages and utilizing Java collections.
- Continue to practice implementing interfaces: you will have to write `fa.nfa.NFA` class that implements `fa.FAInterface` and `fa.nfa.NFAInterface` interfaces.
- Continue to practice extending abstract classes: you will have to write `fa.nfa.NFAState` class that extends `fa.State.java` abstract class.
- Implementing the graph search algorithms: Breadth First Search (BFS) and/or Depth First Search (DFS).

3 Specification

- You are given three packages `fa`, `fa.dfa` and `fa.nfa` and tests folder. Below is the directory structure of the provided files:

```
|-- fa
|    |-- FAInterface.java
|    |-- State.java
|    |-- dfa
|    |    |-- DFAInterface.java
|    |    |-- DFA.java
|    |    |-- DFAState.java
|    |-- nfa
|         |-- NFADriver.java
|         |-- NFAInterface.java
|-- tests
        |-- p2tc0.txt
        |-- p2tc1.txt
        |-- p2tc2.txt
        |-- p2tc3.txt
```

To compile `fa.nfa.NFADriver` from the top directory of these files:

```
[you@onyx]$ javac fa/nfa/NFADriver.java
```

To run `fa.nfa.NFADriver`:

```
[you@onyx]$ java fa.nfa.NFADriver ./tests/p2tc0.txt
```

- You will use all classes in these packages.
- Classes that you will have to create in `fa.nfa` package are `NFA.java` and `NFAState.java`.

3.1 Input file to `NFADriver`

The input file has the following format:

- The 1st line contains the names/labels of the final states, i.e., F. The names are separated by the white space. The first line can be empty if there is no final states.
- The 2nd line contains the name/label of the start state, i.e., q_0.
- The 3rd line contains the rest of states, i.e., those states that are neither in F nor q_0. Note that this line can be an empty line.
- The 4th line lists the transitions. Transitions are separated by the white space. Three symbols compose a transition $s_0 s_1 s_2$, where

  - s_0 is the name of the "from" state.
  - s_1 is the symbol from the alphabet, i.e., $s_1 \in \Sigma$ or the empty string ε for which we use 'e' symbol, that is 'e' $\notin \Sigma$.
  - s_2 is the name of the "to" state.

- Starting from line 5, each line contains a single string for which we need to determine whether it is in the language of the DFA. The strings are over the alphabet and we use 'e' symbol to represent the empty string ε.

For example this NFA
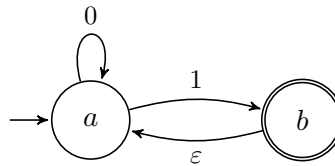
has the following encoding and the set of strings to test:

```
b
a

a0a a1b bea
0
1
00
101
e
```

3.2 `fa` package

This package contains an abstract class and an interface that describe common features and behaviors of a finite automaton.

3.3 `fa.dfa` package

This package contains an implementation of a DFA that `fa.nfa.NFASimulator` class uses and your implementation of `fa.nfa.NFA` class will use to construct an equivalent DFA. Please take a minute to look at the code and notice what data-structures are used and how the transition function is implemented. We hope it will serve you as a good example for choosing the proper data-structures for your NFA implementation.

3.4 `fa.nfa.NFASimulator` (provided class, the driver class)

In `fa.nfa` package you are given a class named `NFASimulator` that reads the input file, instantiates a corresponding NFA object, obtains an equivalent DFA and simulates that DFA instance on each input string. `NFASimulator.java` is adequately documented. This class takes a test case file as an argument and produces the following output: the description of the file machine (by calling `toString()` method of `DFA` class), a blank line and the answer whether an input string is in the language of the machine, i.e., if the DFA accepts that string. Refer to Sample Input/Output section for examples.

3.5 `fa.nfa.NFA` (class you need to implement)

In `nfa` package `NFA` class *must* implement `fa.FAInterface` and `fa.nfa.NFAInterface` interfaces. Make sure to implement all methods inherited from those interfaces. You have to add instance variables representing NFA elements. You can also write additional methods which must be private, i.e., only helper methods.

The method `public DFA getDFA()` is where **you will spend most of your time**. Each DFA state will correspond to a **set** of NFA states. You should track inside `getDFA()` method whether a DFA state with the label corresponding to the string representation of the NFA

states have been created or not. The order in which NFA states appear in DFA's label does not matter.

The implementation of `public DFA getDFA()` will require walking through an NFA, i.e., graph traversing. You can do that by either using depths-first search (DFS) or breadth-first search (BFS) algorithms that you've studied in CS 321. Depending on what type of search you will choose the ordering of DFA's label might be different. In the implementation that we used to demonstrate the expected output we use BFS implementation.

The method `public Set<NFAState> eClosure(NFAState s)`, i.e., the epsilon closure function, computes the set of NFA states that can be reached from the argument state `s` by going only along ε transitions, including `s` itself.

Note: We encourage you to implement `eClosure` first and call it from `getDFA` method.

3.6 `fa.nfa.NFAState` (class you need to implement)

In `fa.nfa` package `NFAState` class *must* extend `fa.State` abstract class. If you wish you can add additional instance variables and methods to your `NFAState` class.

4 Sample Input/Output

Below is the sample input/output for four test cases provided to you, p2t0.txt is not part of the test suite on which you will be graded, but the rest three are.

```
[you@onyx p2]$ cat p2t0.txt
b
a

a0a a1b bea
0
1
00
101
e
[you@onyx p2]$ java fa.nfa.NFASimulator p2t0.txt
Q = { [a] [a, b] }
Sigma = { 0 1 }
delta =
                  0          1
       [a]        [a]      [a, b]
    [a, b]        [a]      [a, b]
q0 = [a]
F = { [a, b] }

no
```

```
yes
no
yes
no
```

```
[you@onyx p2]$ cat p2t1.txt
2
1

1a1 1a2 1b2
a
aa
b
bb
aba
[you@onyx p2]$ java fa.nfa.NFASimulator p2t1.txt
Q = { [1] [1, 2] [2] [] }
Sigma = { a b }
delta =
                      a            b
         [1]      [1, 2]          [2]
      [1, 2]      [1, 2]          [2]
         [2]          []           []
          []          []           []
q0 = [1]
F = { [1, 2] [2] }

yes
yes
yes
no
no
```

```
[you@onyx p2]$cat p2t2.txt
2
0
1
0a0 0b0 0a1 0b2 1a0 1b1 2a0 2b0
a
aa
b
bb
aba
[you@onyx p2]$ java NFASimulator p2t2.txt
Q = { [0] [1, 0] [0, 2] [1, 0, 2] }
Sigma = { a b }
```

```
delta =
                       a          b
           [0]     [1, 0]     [0, 2]
        [1, 0]     [1, 0] [1, 0, 2]
        [0, 2]     [1, 0]     [0, 2]
     [1, 0, 2]     [1, 0] [1, 0, 2]
q0 = [0]
F = { [0, 2] [1, 0, 2] }

no
no
yes
yes
no
```

```
[you@onyx p2]$cat p2t3.txt
s
q
r
q0s q1r res r0q ser s1q
101
0001
100
11
110
[you@onyx p2]$ java fa.nfa.NFASimulator p2t3.txt
Q = { [q] [r, s] }
Sigma = { 0 1 }
delta =
                       0          1
           [q]     [r, s]     [r, s]
        [r, s]        [q]        [q]
q0 = [q]
F = { [r, s] }

yes
no
yes
no
yes
```

5 Grading Rubric

1. 5 points for the properly formatted and written README.
2. 5 points for the proper code documentation: Javadocs and inline comments.

3. 5 for correct program submission and program compiling/running on onyx.

4. 10 points for the code quality, i.e., easy to read, proper data structures used and proper variable naming, proper object-oriented design.

5. 75 points for program running correctly. We will have 15 test files (3 of which are provided to you) each containing 5 test input strings. For each correctly accepted/rejected string you will get 1 point. So, if all test files pass then you will get $15 \times 5 = 75$.

6   Submitting Project 1 Part 2

Documentation:
If you haven't done it already, add **Javadoc comments** to your program. It should be located immediately before the class header and after each method

- Have a class javadoc comment before the class.
- Your class comment must include the *@author* tag at the end of the comment. This will list you as the author of your software when you create your documentation.
- Use *@param* and *@return* tags. Use inline comments to describe how you've implemented methods and to describe all your instance variables.

Include a plain-text file called **README** that describes your program and how to compile and run it. Expected formatting and content are described in README_TEMPLATE. An example is available in README_EXAMPLE.
You will follow the same process for submitting each project.

1. Open a console and navigate to the project directory containing your source files,

2. Remove all the `.class` files.

3. In the same directory, execute the submit command :
   ```
   submit cs361 cs361 p1p2
   ```

4. Look for the success message and timestamp. If you don't see a success message and timestamp, make sure the submit command you used is EXACTLY as shown

Required Source Files:
Make sure the names match what is here **exactly** and are submitted in **the proper folders/packages**

- `NFA.java` in `fa.nfa` package.
- `NFAState.java` in `fa.nfa` package.
- `README`.

After submitting, you may check your submission using the "check" command. In the example below:
```
submit -check cs361 cs361 p1p2
```