

CS 361: Project 2 (Nondeterministic Finite Automata)

Due Nov. 18, 2022
Fall 2022

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 to a NFA's instance.

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.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 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
```

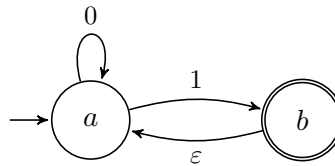
- You will use all classes in these packages.
- Classes that you will have to create in `fa.nfa` package are `NFA.java` and `NFASState.java`.

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 are 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 q_0 nor in F . 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 $s_0s_1s_2$, 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

Note: The 3rd line is empty because both states are already specified on 1st and 2nd lines. We will give you four test files (three from the test suite you will be graded on), but we encourage you to create several of your own.

3.2 fa package

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

Note: You must not modify classes in this package.

3.3 fa.dfa package

This package contains the instructor's implementation of a DFA that `fa.nfa.NFADriver` 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, you will use it as an example for choosing proper data-structures for your NFA implementation.

Note: You must not modify classes in this package. That is, you cannot add/change other methods, e.g., override `equals` in `DFA.java` and `DFAState.java`.

3.4 fa.nfa.NFADriver (provided class, the driver class)

Note: You don't need to modify the class. You will use it to test your NFA implementation. In `fa.nfa` package you are given a class named `NFADriver` that reads the input file, instantiates a corresponding NFA object, obtains an equivalent DFA and simulates that DFA instance on each input string. `NFADriver.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.nfa.NFAInterface` interface. 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.

Note: Use correct data-structures for each of NFA's element, e.g., set of states should be modeled by a class implementing `java.util.Set` interface.

The method `public DFA getDFA()` is where **you will spend most of your development time**. Recall, that each DFA state corresponds to a **set** of NFA states. You should track inside `getDFA()` method whether a DFA state with the label(name) corresponding to the string representation of the NFA states has been created or not. There is no requirements

on the order in which NFA states appear in a DFA's label. (In p2t0.txt example the states Q of a DFA can be printed either as $\{[a] \ [a, b]\}$ or as $\{[b, a] \ [a] \}$, where a, b are the states of the corresponding NFA.)

The implementation of `public DFA getDFA()` will require walking through an NFA, i.e., traversing a graph. To do so, you must implement **the breadth-first search (BFS)** algorithm (a loop iterating over a queue; an element of a queue is a set of NFA states).

The method `public Set<NFASate> eClosure(NFASate 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. You must implement it using **the depth-first search algorithm (DFS)** using a recursion, i.e., `eClosure` should invoke itself or another helper method, e.g., `private Set<NFASate> eClosure(NFASate s, Set<NFASate> visited)` that invokes itself.

Note: We strongly encourage you to implement `eClosure` first and then implement `getDFA` method that calls `eClosure`.

3.6 fa.nfa.NFASate (class you need to implement)

In `fa.nfa` package `NFASate` class *must* extend `fa.State` abstract class. If your implementation requires it, you can add additional instance variables and methods to your `NFASate` 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.NFADriver 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.NFADriver 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 fa.nfa.NFADriver 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.NFADriver 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. 3 points for the properly formatted and written README.
2. 3 points for the proper code documentation: Javadocs and inline comments.
3. 4 for correct program submission and program compiling/running on onyx.
4. 5 for DFS in `eClosure` using `recursion`.
5. 5 for BFS in `getDFA`.
6. 5 points for code quality, i.e., easy to read, proper data structures used and proper variable naming, proper object-oriented design, instance variables are private and initialized in constructors.
7. 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 before each method that was not inherited

- 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 each team member 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](#). Once again, make sure to list all team members in the README file, along with the class section you belong to. Make sure it is the section you are enrolled in, not the one you attend. This also goes for submitting the project, be sure the project is submitted into **only one** of the section a group member is enrolled in. This will make it easier for TAs to find you on class rosters across sections. They will definitely appreciate it =)

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 :

Section 1:



Section 2: `submit cs361 cs361 p2_002`




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 structures for packages**. Please only submit the files listed here, any other files submitted will be ignored.

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

After submitting, you may check your submission using the “check” command. Like in the example below, where X is your section:



```
submit -check cs361 cs361 p2_002
```