CS 361: Project 2 (Nondeterministic Finite Automata)

Fall 2024
Due Date: 10/28/24

1 Project Overview

In this project, you will implement two classes that model an instance of a nondeterministic finite automaton (NFA) and its behavior.

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 Where to start in test-driven development.

As in the previous assignment, you are given an abstract class and interfaces, as well JUnit test file. In test-driven development, first we add declarations of all required/public methods, i.e., just methods' skeletons. Then we run the JUnit file, where all test cases initially fail. As we continue to implement methods' functionalities, our program passes more and more test cases, until all of them are passed :) .

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

```
|-- fa
|    |-- FAInterface.java
|    |-- State.java
|    |-- nfa
|          |-- NFAInterface.java
|-- test
       |-- nfa
            |-- NFATest.java
```

- You will use all classes in these packages.
- Start the project by creating in `fa.nfa` package `NFA.java` that implements `NFAInterface.java`, and `NFAState.java` that extend `State.java`. Then add declarations for all unimplemented methods (IDE can do it for you automatically). Even though the methods are not implemented, now they can be compiled and run.

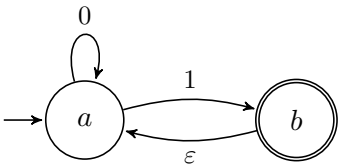
- To compile `test.nfa.NFATest` on **onyx** from the top directory of these files:
```
[you@onyx]$ javac -cp .:/usr/share/java/junit.jar ./test/nfa/NFATest.java
```
To run `test.nfa.NFATest` on **onyx** type in one single line:
```
[you@onyx]$ java -cp .:/usr/share/java/junit.jar:/usr/share/java/hamcrest/core.jar
org.junit.runner.JUnitCore test.nfa.NFATest
```
- Continue to implement by adding more functionalities into the methods until all test cases are passing (add a few more of your own).

4 Specifications

- Existing classes that you will use: `test.nfa.NFATest`, `fa.nfa.NFAInterface`, `fa.FAInterface`, `fa.State`.
- Classes that you will implement and submit: `fa.nfa.NFA` and `fa.nfa.NFAState`.

4.1 Creating an NFA

JUnit uses a series of invocations to create an NFA object. For example, to create the following NFA



It uses the following method invocations (with assertions) on the created NFA object:
```
NFA nfa = new NFA();

nfa.addSigma('0');
nfa.addSigma('1');

assertTrue(nfa.addState("a"));
assertTrue(nfa.setStart("a"));

assertTrue(nfa.addState("b"));
assertTrue(nfa.setFinal("b"));

assertTrue(nfa.addTransition("a", Set.of("a"), '0'));
assertTrue(nfa.addTransition("a", Set.of("b"), '1'));
assertTrue(nfa.addTransition("b", Set.of("a"), 'e'));
```

NFA labels can have arbitrary length; 'e' character is reserved for marking ε transitions and is never part of the alphabet.
We provide you with tests for three NFA encodings, but we encourage you to create several of your own to further test your implementation.

4.2 `NFATest` given class in `test.nfa` package, this is the driver class.

Note: You should not modify the existing methods. You will use them to test your NFA implementation.

You are given `NFATest`[1] class that instantiates an NFA and checks for expected behavior. For each NFA instance, it has a series of 6 tests (X is a NFA id, e.g., 1, 2 and so on):

- `test1_X` invokes an NFA instantiation method and checks whether adding, setting states and transition methods of NFA work as expected.
- `test2_X` checks the correctness of the successfully instantiated NFA, e.g., states have correct names and types.
- `test3_X` determines if the NFA is actually a DFA.
- `test4_X` invokes `eClousre` of an NFA state and checks if it computes correct sets of NFA states
- `test5_X` passes strings to the NFA and checks whether it accepts strings in the NFA's language and rejects string not in the NFA's language.
- `test6_X` passes strings to the NFA and checks for the maximum number of NFA copies.

Your implementation should pass all test cases of `NFATest`. During grading, we add 11 more similar test cases to further test your implementation. Thus, we encourage you to create additional test cases, too.

4.3 `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.

4.4 `NFA` class you need to implement in `fa.nfa` package

In `nfa` package, `NFA` class *must* implement `fa.nfa.NFAInterface` interface. Make sure to implement all methods inherited from that interface. You should 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.

Below are additional requirements:

[1] We omit the package in the class name for brevity.

1. The method `public boolean accepts(String input)` is where **you more likely spend most of your development time**. It requires walking through an NFA, i.e., traversing a graph using BFS, to keep track of all the NFA's copies created while following symbols in `input`.
   For example, the `accepts` method of the above NFA for the input string "011" needs to simulate the following multi-copy configuration trace:

   $$(\{a\}, 011) \vdash (\{a\}, 11) \vdash (\{a, b\}, 1) \vdash (\{a, b\}, \varepsilon)$$

   Where the copies are inside {} follow along the read symbols from one state to another. In the above example, at the beginning, there is only one copy in the start state a, after reading 0, there is only one copy in the same state 0. Next, after reading the first 1, NFA creates two copies: one in state b (resulting from transitioning from a on 1), and another in state a (result of spanning a copy from b on ε transition). Copies that end up in the same state after a transition or closure are considered to be identical and merged into one copy.

   The method returns `false` when after reading all the symbols, none of the copies are in any accepting state (that includes when no alive copies remain).

2. The method `public int maxCopies(String input)` does similar simulation as `accepts`, only it returns the maximum number of copies a trace can generate. In the case of the above example, the method returns 2 since the maximum number of copies in the trace is $\{a, b\}$ and $|\{a, b\}| = 2$.

3. 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 following only along ε transitions, including `s` itself. You must implement it using **the depth-first search algorithm (DFS)** using a stack in a loop, i.e., `eClosure`'s loop should push children of the current node on the stack, but only those children that are reachable through an ε transition, for which NFA uses reserved character 'e'.
   Note: We strongly encourage you to implement `eClosure` first and then implement `accepts` method that calls `eClosure`.

4. The method `public boolean isDFA()` determines if an NFA is actually a DFA, i.e., all of its transitions obey the rule's of DFA transitions.

4.5 `NFAState` class you need to implement in `fa.nfa` package

In `fa.nfa` package `NFAState` class *must* extend `fa.State` abstract class. We recommend that you add an instance variable to model the state's transitions with the following map `Map<Character, Set<NFAState>>`. If your implementation requires it, you can add additional instance variables and methods to your `NFAState` class.

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. **3** for correct program submission and the program compiling/running on onyx.
4. **4** for DFS in `eClosure` using `a stack and a loop`.
5. **3** 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.
6. **84** for passing tests cases. We will have 14 test NFAs (3 of which are provided to you) each containing 6 test cases. For each correctly passed test, you will get 1 point. So, if all test cases pass, you will receive $14\times\ 6 = 84$.

6 Submitting Project 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 the members of your team as the authors of the software when you create its 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 the program and how to compile and run it. Remember to include also in the README who are the authors of the code. If you are submitting as a team, then only one team member should submit the project to prevent duplicate grading. Expected formatting and content are described in README_TEMPLATE. An example is available in README_EXAMPLE. **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`.

Submission instructions:
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:
   ```
   submit cs361 cs361 p2_1
   ```
   Section 2:
   ```
   submit cs361 cs361 p2_2
   ```

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
5. 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_X
   ```