Course Project 1
Nondeterministic Finite Automata

CSE 30151 Spring 2018

Version of 2018/01/19

We've studied the theory of nondeterministic finite automata, and now it's time to implement them. The interesting challenge is that NFAs are nondeterministic, but real computers are deterministic – how do we simulate nondeterminism?

One option is backtracking: when two transitions are possible, try one, and if it fails, try the other. But this will lead to a $O(2^n)$ time algorithm (where n is the input length). The theory provides another option: convert the NFA to an equivalent DFA. That gives a $O(n)$ algorithm, but the conversion could take $O(2^{|Q|})$ time and space (where $|Q|$ is the number of states).

In this project, you'll implement a solution that runs in $O(|\delta|n)$ time, where $|\delta|$ is the number of transitions. You can write your implementation in C++ or Python (or another language with permission of the instructor).

Getting started

You should have been given access on GitHub to a repository called `theory-project-`*`team`*, with *`team`* replaced by your team's name. Please clone this repository to wherever you plan to work on the project:

```
git clone https://github.com/ND-CSE-30151-SP18/theory-project-team
cd theory-project-team
```

If you're the first team member to do this, your repository is empty. In that case, run the commands:

```
git pull https://github.com/ND-CSE-30151-SP18/theory-project-skeleton
git push
```

If one of your teammates already did this, there's no need for you to repeat it. Whenever we make an update to `theory-project-skeleton`, we'll send out an announcement, and one of you will need to repeat the pull/push (resolving any merge conflicts if necessary) to get the update.

Now your directory should include the following files (among others):

```
bin.{linux,darwin}/
  run_nfa
  union_nfa
  concat_nfa
  star_nfa
  re_to_nfa
  compare_nfa
examples/
  sipser-n1.nfa
  sipser-n2.nfa
  sipser-n3.nfa
  sipser-n4.nfa
tests/
  test-cp1.sh
cp1/
```

- The `bin.linux` and `bin.darwin` contain binaries for Linux and Mac, respectively. They contain reference implementations for the tools you will implement and tools used by the test scripts.
- The `examples` directory contains examples of NFAs that you will use for testing. See below for a description of the file format.
- The `tests` directory contains test scripts. The script `tests/test-cp1.sh` tests your code for correctness and speed. Your code needs to pass all tests in order to get full credit.
- Please place the programs that you write into the `cp1/` subdirectory.

1 Reading and writing

Write a function to read an NFA from a file:

- Argument: name of file containing definition of NFA M
- Return: an NFA M

And write a function to write an NFA to a file:

- Arguments: NFA M and name of file to write to
- Effect: Definition of M is written to file

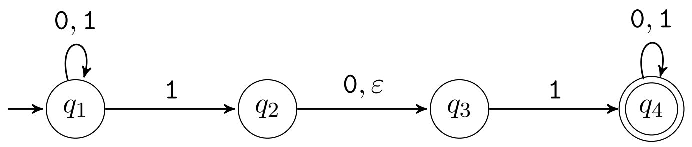
The NFA definition should have the following format. It should begin with a four-line header:

1. A whitespace-separated list of states, Q.
2. A whitespace-separated list of input symbols, Σ. It should be disjoint from Q.
3. The start state, $s \in Q$.
4. A whitespace-separated list of accept states, $F \subseteq Q$.

The rest of the lines list the transitions, one transition per line. Each line has three fields, separated by whitespace:

1. The state $q \in Q$ that the transition leaves from.
2. The symbol $a \in \Sigma$ that the transition reads, or `&` for the empty string.
3. The state $r \in Q$ that the transition goes to.

For example, the following NFA (N_1 in the book):



would be specified by the file (`examples/sipser-n1.nfa`):

```
q1 q2 q3 q4
0 1
q1
q4
q1 0 q1
q1 1 q1
q1 1 q2
q2 0 q3
q2 & q3
q3 1 q4
q4 0 q4
q4 1 q4
```

There are no tests for this part.

2 Matcher

Note: This part and part 3 can be done in parallel.

Write a function that tests whether a NFA M accepts a string w:

- Argument: a NFA M
- Argument: a string w
- Return: true iff M accepts w

This function is required to be **linear both in the length of w and the number of transitions in M.**

Define a *configuration* of M on input string w to be a pair (q, i), where $q \in Q$ and $0 \le i \le |w|$. These configurations can be thought of as nodes in a graph. For example, if the NFA is N_1 above and $w =$ `010110`, then the graph of configurations would be:

$$\begin{tikzcd}
{} \arrow[d] & & & \\
q_1, 0 \arrow[d] & & & \\
q_1, 1 \arrow[d] \arrow[dr] & & & \\
q_1, 2 \arrow[d] & q_2, 2 \arrow[r] \arrow[dr] & q_3, 2 & \\
q_1, 3 \arrow[d] \arrow[dr] & & q_3, 3 \arrow[dr] & \\
q_1, 4 \arrow[d] \arrow[dr] & q_2, 4 \arrow[r] & q_3, 4 \arrow[dr] & q_4, 4 \arrow[d] \\
q_1, 5 \arrow[d] & q_2, 5 \arrow[r] \arrow[dr] & q_3, 5 & q_4, 5 \arrow[d] \\
q_1, 6 & & q_3, 6 & \boxed{q_4, 6}
\end{tikzcd}$$

This is similar to Sipser's Figure 1.29, but there are several differences here. The most important difference is that configuration $(q_4, 5)$ appears only once with two incoming edges, instead of appearing twice. In general, each configuration appears at most once in the graph. As a result, the graph has at most $|Q||w| + 1$ nodes and $|\delta||w|$ edges.

Deciding whether N_1 accepts w amounts to searching for a path from the start configuration (in this case, $(q_1, 0)$) to an accept configuration (in this case, $(q_4, 6)$). You can use any graph search algorithm, but in later projects it will turn out that depth-first search corresponds most closely to how the real Unix tools work.

Package the above into a command-line tool called `run_nfa`:

`./run_nfa` *nfafile*

where *nfafile* is a file defining an NFA. The program should read zero or more lines from stdin and write to stdout just the lines that are accepted by the NFA. Test your program by running `test-cp1.sh`. This script runs `run_nfa` on several NFAs and several test strings, and it also produces a graph of the running time of `run_nfa` on NFAs of various sizes. The sizes are chosen so that the graph should look roughly linear, like this:

```
n= 32  *
n= 45   *
n= 55   *
n= 64    *
n= 71    *
n= 78    *
n= 84     *
n= 90     *
n= 95      *
n=100      *
```

3 Regular operations

Write functions that perform the regular operations, using the constructions given in the book:

union(M_1, M_2)

- Arguments: NFAs M_1, M_2
- Returns: NFA recognizing language $L(M_1) \cup L(M_2)$

concat(M_1, M_2)

- Arguments: NFAs M_1, M_2
- Returns: NFA recognizing language $L(M_1) \circ L(M_2)$

star(M)

- Argument: NFA M
- Returns: NFA recognizing language $L(M)^*$

Write three programs, called `union_nfa`, `concat_nfa`, and `star_nfa`, to test your operations. Each takes one or two command-line arguments, each of which is the name of a file containing a NFA, in the same format you used in CP1, and each writes a NFA to stdout in the same format.

`union_nfa` *nfafile1 nfafile2*	Writes union of NFAs to stdout
`concat_nfa` *nfafile1 nfafile2*	Writes concatenation of NFAs to stdout
`star_nfa` *nfafile*	Writes Kleene star of NFA to stdout

Test your programs by running `test-cp1.sh`.

Submission instructions

Your code should build and run on `student`*nn*`.cse.nd.edu`. The automatic tester will clone your repository, run `make -C cp1`, and then run `tests/test-cp1.sh`. You're advised to try all of the above steps and ensure that all tests pass.

To submit your work, please push your repository to Github and then create a new release with tag version `cp1` (note that the tag version is not the same thing as the release title). If you are making a partial submission, then use tag version `cp1-1` for part 1, `cp1-2` for part 2, and so on.

Rubric

Reader	3
Writer	3
Matcher	9
`run_nfa`	
correctness	3
time complexity	3
Union	3
Concatenation	3
Kleene star	3
Total	30