Course Project 1
Nondeterministic Finite Automata

CSE 30151 Spring 2018

Version of 2018/01/25
Due 2018/02/08 22:00

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}/
  nfa_path
  re_to_nfa
examples/
  sipser-n1.nfa
  sipser-n2.nfa
  sipser-n3.nfa
  sipser-n4.nfa
  cycle.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 NFAs

Design a data structure for representing a NFA M, and write functions to read and write NFAs. (For all projects, the names of functions and the way that they are called are just suggestions; if you prefer a different style, that's fine.)

read_nfa(*filename*)

- *filename*: Name of file containing definition of NFA M
- Returns: The NFA M

write_nfa(M, *filename*)

- *M*: The NFA to write
- *filename*: Name of file to write to
- Effect: Writes definition of M to file

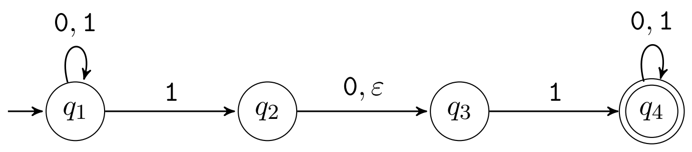
NFA file format The NFA definition should have the following format. It should begin with a four-line header:

1. A whitespace-separated list of states, Q.
2. A whitespace-separated list of input symbols, Σ. It should be disjoint from Q. Each symbol should be exactly one character long.
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
```

2 Matcher

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

match(M, w)

- M: An NFA to run
- w: The string to run on
- Returns: A pair (*flag*, *path*), where
  - *flag*: True if M accepts w; false otherwise.
  - *path*: List of the transitions on an accepting path. If there is more than one, an arbitrary path is returned.

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

Here's how to do this. Define a *configuration* of M on input string w to be a pair (q, i), where $q \in Q$ and $0 \leq i \leq |w|$. 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:

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

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.

Then, 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; **breadth-first search** is probably the least hassle, but **depth-first search** corresponds to how the real Unix tools work. (One particular thing to watch out for is cycles of ε-transitions, as in `cycle.nfa`. Make sure your matcher doesn't hang when it encounters one.)

When you studied graph search algorithms, you may not have seen how to reconstruct the found path. Graph searches maintain a set that keeps track of which configurations have been visited (in your Data Structures notes, it was called `marked`). If you change this to a data structure that records, for each configuration, how you got to that configuration, then after the search finishes, you can use that information to reconstruct the path.

3 Putting it together

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

`nfa_path` *nfafile string*

- *nfafile*: name of file defining an NFA M
- *string*: string to run M on
- Output:
  - If M accepts *string*, prints `accept` followed by an accepting path
  - Otherwise, prints `reject`

The path should be printed with one transition per line, in the same format as the NFA file format. For example:

```
$ nfa_path examples/sipser-m1.nfa 11
accept
q1 1 q2
q2 & q3
q3 1 q4
```

Test your program by running `test-cp1.sh`. This script runs `nfa_path` on several NFAs and several test strings, and it also produces a graph of the running time of `nfa_path` 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      *
```

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

Part 1	
data structure	3
`read_nfa`	3
`write_nfa`	3
Part 2 (`match`)	
correct algorithm	6
handling ε	3
reconstructing path	6
Part 3 (`nfa_path`)	
correctness	3
time complexity	3
Total	30