Course Project 1
Nondeterministic Finite Automata

CSE 30151

Spring 2017

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 = 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.

It turns out that a $O(|Q|n)$ time solution is possible, and this is the solution you'll implement in this project.[1] It has three steps. First, read in NFA M and string w, and construct an automaton M_w that recognizes the language $\{w\}$. Second, intersect M and M_w. Third, check whether the intersection is empty. Then M accepts w iff $L(M \cap M_w) \neq \emptyset$.

Getting started

You should have access on GitHub to a repository called `theory-project-`*`team`*. Please clone this repository to wherever you plan to work on the project. The repository includes the following files:

```
bin/
  compare_nfa
  singleton_nfa
  empty_nfa
  intersect_nfa
  run_nfa
examples/
  empty1.nfa
  empty2.nfa
```

[1] It's not the space-optimal solution, but it factors in a way that will come in handy later.

```
  nonempty1.nfa
  sipser-n1.nfa
  sipser-n2.nfa
  sipser-n3.nfa
  sipser-n4.nfa
tests/
  test-cp1.sh
doc/
  cp1.pdf
cp1/
```

Please place the programs that you write into the `cp1/` subdirectory.

1 Data structure

Design a data structure for representing a NFA M. See the emptiness and intersection operations below to get an idea of how it will be used. These operations are simplest if the data structure supports the following operations:

- Add a new transition $q \xrightarrow{a} r$.
- Iterate over all states.
- Iterate over all input symbols.
- Iterate over transitions out of state q.
- Iterate over transitions on input symbol a.

2 Reading and writing

Write a function to read an NFA from a file:

- Input: name of file containing definition of NFA M
- Output: (data structure representing) M

And a function to write an NFA to a file:

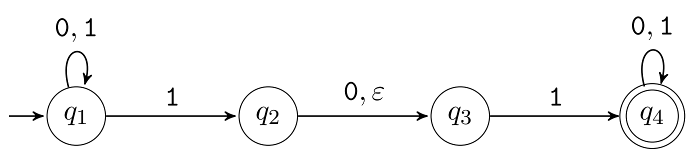
- Input: NFA M and name of file to write to
- Result: Definition of M is written to file

The NFA definition should have the following format.

- Line 1 lists all the states in Q, separated by whitespace.
- Line 2 lists all the alphabet symbols in Σ, separated by whitespace.
- Line 3 is the start state, s.
- Line 4 lists all the accept states in F, separated by whitespace.
- The rest of the lines list the transitions, one transition per line. Each line has three fields, separated by whitespace:
  - a state q
  - a symbol $a \in \Sigma$, or $a = \texttt{\&}$ for the empty string[2]
  - a state in $\delta(q, a)$.

A `//` anywhere in a line introduces a comment that extends to the end of the line.[3]

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



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

```
q1 q2 q3 q4 // states
0 1         // alphabet
q1          // start
q4          // accept
q1 0 q1
q1 1 q1
q1 1 q2
q2 0 q3
q2 & q3     // & for epsilon
q3 1 q4
q4 0 q4
q4 1 q4
```

[2]An ampersand is a stylized *et*, and ε is a Greek *e*, so it kind of makes sense.

[3]The more common `#` isn't ideal because this character is often used as an alphabet symbol.

3 Singleton

Note: Parts 3, 4 and 5 can be written and tested independently.

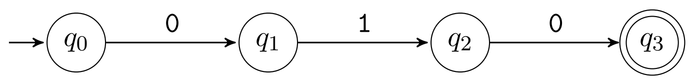
Write a function that constructs an automaton that accepts a single string.

- Input: string w
- Output: NFA M such that $L(M) = \{w\}$

If $|w| = n$, this automaton has $(n+1)$ states $q_0, \ldots, q_n$, with q_0 the start state and q_n the accept state, and

$$\delta(q_i, a) = \begin{cases} \{q_{i+1}\} & \text{if } a = w_{i+1} \\ \emptyset & \text{otherwise.} \end{cases}$$

For example, if $w = \texttt{010}$, then the corresponding automaton would be



Write a program to test your function:

`./singleton_nfa` w

should write the NFA that recognizes $\{w\}$ to stdout. Test your program by running `test-cp1.sh`.

4 Emptiness

Write a function that tests whether an NFA recognizes the empty language.

- Input: NFA M
- Output: true iff $L(M) = \emptyset$

Use breadth-first search to check whether there is any path from the start state to an accept state, and return true iff there is none. (Why shouldn't you use depth-first search?)

Write a program to test your function:

`./empty_nfa` *nfafile*

should exit with status 0 if the NFA recognizes the empty language, and 1 otherwise. Test your program by running `test-cp1.sh`.

5 Intersection

Write a function to compute the intersection of two NFAs.

- Input: NFAs M_1, M_2
- Output: NFA $M_1 \cap M_2$

The construction for DFAs is given a brief mention in the book (page 46, footnote 3). For NFAs with epsilon transitions, it goes like this. Given

$$M_1 = (Q_1, \Sigma, s_1, F_1, \delta_1)$$
$$M_2 = (Q_2, \Sigma, s_2, F_2, \delta_2)$$

let

$$M = (Q, \Sigma, s, F, \delta)$$

where

$$Q = Q_1 \times Q_2$$
$$s = (s_1, s_2)$$
$$F = F_1 \times F_2$$

and δ is defined as follows:

- For all $q_1, q_2 \in Q$, $a \in \Sigma$, if $r_1 \in \delta_1(q_1, a)$ and $r_2 \in \delta_2(q_2, a)$, then $(r_1, r_2) \in \delta((q_1, q_2), a)$.
- For all q_1, q_2, if $r_1 \in \delta_1(q_1, \varepsilon)$, then $(r_1, q_2) \in \delta((q_1, q_2), \varepsilon)$.
- For all q_1, q_2, if $r_2 \in \delta_2(q_2, \varepsilon)$, then $(q_1, r_2) \in \delta((q_1, q_2), \varepsilon)$.
- Nothing else is in δ.

Write a program to test your function:

`./intersect_nfa` *nfafile nfafile*

should read two NFAs and write their intersection to stdout. Test your program by running `test-cp1.sh`.

6 Put it all together

Put all of the above pieces together to write a function:

- Input: NFA M, string w
- Output: true iff M accepts w.

Package your function in a command-line tool:

`./run_nfa` *nfafile*

where *nfafile* is a file defining an NFA. The program should read lines from stdin and write to stdout just the lines that are accepted by the NFA. Test your program 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` in subdirectory `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 `cp1`. If you need to re-submit, create another release with a new tag starting with `cp1`, like `cp1.1`.

Rubric

Data structure	3
Reader	3
Writer	3
Singleton	3
Emptiness	6
Intersection	6
Main program	6
Total	30