Course Project 1
Nondeterministic Finite Automata

CSE 30151 Spring 2025

Version of 2024-12-31
Due date 2025-02-07

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 third solution, one that runs in $O(|\delta|n)$ time, where $|\delta|$ is the number of transitions. You can write your implementation in Python, or another language with permission of the instructor.

Getting started

1. Visit this link to accept the assignment in GitHub Classroom: https://classroom.github.com/a/xkexqxUm.
2. You should be given the choice either to create a new team or to join an existing team.
3. A repository will be created for you called `theory-project-`*`team-name`*, where *`team-name`* is replaced with your team's name.
4. Clone this repository to wherever you plan to work on the project:

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

Your directory should include the following files (among others):

```
bin.{linux,darwin}/
  nfa_path
  re_to_nfa
  epsilon_nfa
  symbol_nfa
examples/
  cycle.nfa
  sipser-n1.nfa
```

```
  ...
tests/
  test-cp1.sh
cp1/
```

- The `bin.linux/` and `bin.darwin/` directories contain binaries for Linux and Mac, respectively. They contain reference implementations for the tools you will implement and tools used by the test scripts.
  - Below, we'll assume that these binaries are in the `bin/` subdirectory, so for convenience you might want to do one of

    ```
    ln -s bin.linux bin  # in Linux
    ln -s bin.darwin bin # on a Mac
    ```

  - On a Mac, if a security warning pops up when you try to run these, then do

    ```
    xattr -dr com.apple.quarantine bin.darwin/*
    ```

- 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.
- If we ever need to make any updates, you'll receive a notification of a pull request, which you should merge.

1 NFAs

Let A be the set of all printable, non-whitespace ASCII characters (to be precise, Unicode code points 33 to 126 inclusive).

1.1 Data structure

Design a data structure for representing a NFA M. Look at Part 1.2 to see what information the data structure needs to include, and look at Part 2 to get an idea of what operations the data structure will need to support efficiently.

1.2 Reading and writing

Write the following 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(*file*)

- *file*: File containing definition of NFA M
- Returns: The NFA M

write_nfa(M, *file*)

- M: The NFA to write
- `file`: File to write to
- Effect: Writes definition of M to file

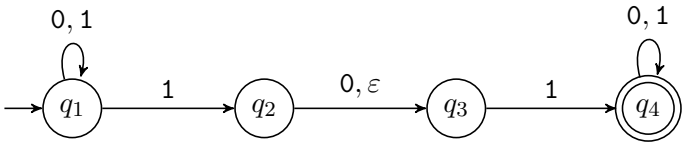
NFA file format The NFA definition must have the following format. It begins with a four-line header:

1. A whitespace-separated list of states, Q. Each state has a name in A^+.
2. A whitespace-separated list of input symbols, Σ, such that $\Sigma \subseteq A \setminus \{\texttt{\&}\}$ and $\Sigma \cap Q = \emptyset$.
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):



is 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
```

1.3 Some simple NFAs

You'll use read_nfa in Part 1.2; to demonstrate write_nfa, write a function that creates an NFA that recognizes the language $\{\varepsilon\}$ and a function that creates an NFA that recognizes the language $\{a\}$ for any $a \in \Sigma$. To test them, write programs with the following command-line usages:

`epsilon_nfa`

- Output: an NFA recognizing the language $\{\varepsilon\}$

`symbol_nfa`

- a: a symbol
- Output: an NFA recognizing the language $\{a\}$

Try running `bin/epsilon_nfa` or `bin/symbol_nfa` if in doubt about what you're supposed to implement.

2 Matcher

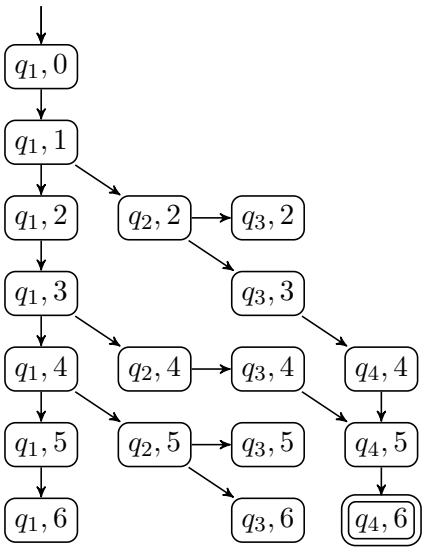
Write a function that tests whether a NFA M accepts a string w:

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 must run in $O(|M| \cdot |w|)$ time, where $|M|$ is the number of states plus 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 $0 \leq i \leq |w|$. The meaning of (q, i) is "M is in state q after reading the first i input symbols." 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 is:



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 twice. In general, each configuration appears at most once in the graph. As a result, the graph has at most $|Q| \cdot |w| + 1$ nodes and $|\delta| \cdot |w|$ 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. Depth-first search is most similar to how the real Unix tools work, but breadth-first search is probably the least hassle. Beware that if you use recursion, you could get an error for exceeding the maximum recursion depth. Another thing to watch out for is cycles of ε-transitions, as in `examples/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. If you change this to a data structure that records, for each configuration, which edge led 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. Your implementation should be at `cp1/nfa_path`; ours is at `bin/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` on one line, followed by an accepting path on subsequent lines
  - Otherwise, prints `reject`

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

```
$ cp1/nfa_path examples/sipser-n1.nfa 11
accept
q1 1 q2
q2 & q3
q3 1 q4
$ cp1/nfa_path examples/sipser-n1.nfa 1
reject
```

You can try running `bin/nfa_path` for comparison.

Test your program by running `tests/test-cp1.sh`. This script compares the output of `cp1/nfa_path` and `bin/nfa_path` on several NFAs and several test strings, and it also produces a graph of the running time of `cp1/nfa_path` on NFAs of various sizes. The sizes are chosen so that if the running time is $\Theta(n^2)$, then the graph will look roughly linear, like this:

```
n=100:      *
n=142:        *
n=174:          *
n=200:             *
n=224:                *
n=245:                  *
n=265:                     *
n=283:                        *
n=300:                          *
n=317:                             *
n=332:                               *
n=347:                                 *
n=361:                                   *
n=375:                                       *
n=388:                                             *
n=400:                                             *
```

Submission instructions

Your code should build and run on `student`*nn*`.cse.nd.edu`. The automatic tester will do the following:

1. Clone your repository and change into its root directory.
2. Run `make -C cp1` (in case you have a Makefile; if you don't, that's fine).
3. 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:

1. Push your repository to GitHub.

2. In GitHub, create a new release by clicking on "Releases," then "Create a new release" or "Draft a new release."

3. Fill in "Release title" with `cp1` if you're submitting the whole assignment, `cp1-1` if you're submitting part 1, `cp1-2` if you're submitting part 2, etc.

4. Click on "Choose a tag," then type the same name you used for the release title, then "Create new tag: `cp1`... on publish."

5. Finally, click "Publish Release."

Rubric

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