Programming Assignment 1
Nondeterministic Finite Automata

Due: Thursday, February 3 at 11:59pm

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 assignment, 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 Python or C++ (or another language with permission of the instructor).

Getting started

You should have been given access on GitHub to a repository named after your team. Please clone this repository to wherever you plan to work on the assignment:

```
git clone https://github.com/ND-CSE-30151-SP22/team.git
cd 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-SP22/theory-project-skeleton.git
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/
  cycle.nfa
  epsilons.nfa
  sipser-n1.nfa
  sipser-n2.nfa
  sipser-n3.nfa
  sipser-n4.nfa
  slow1.nfa
  slow2.nfa
  slow3.nfa
pa1/
  README.md
tests/
  test-pa1.bash
```

- 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-pa1.bash` 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 `pa1/` subdirectory.

1 NFAs

Design a data structure for representing a NFA M, and write functions to read and write NFAs. (For all programming assignments, 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

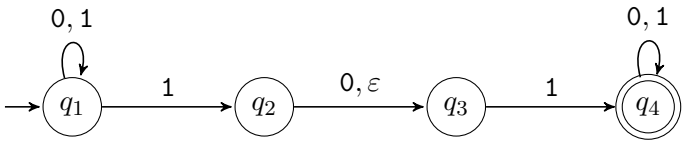
NFA file format The NFA definition should have the following format. It should begin with a four-line header:

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

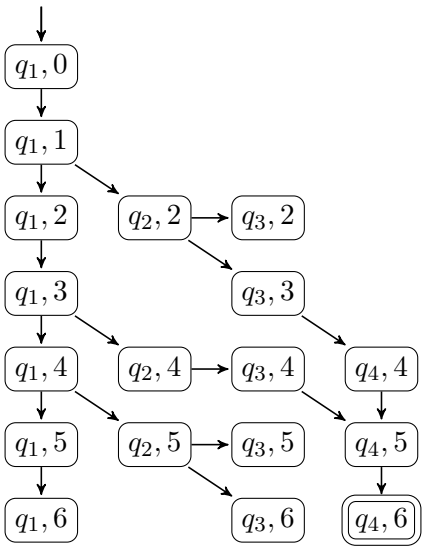
Write a function that tests whether a NFA M accepts a string w:

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:



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, including breadth-first search or depth-first search. One particular thing to watch out for is cycles of ε-transitions, as in `cycle.nfa`. Make sure your graph search algorithm doesn't loop forever when it encounters one.

Some of you may not have yet encountered graph search algorithms in your coursework. Here is the pseudocode for a generic breadth-first search on a graph G:

```
function BREADTH-FIRST-SEARCH(G, r)  ▷ Let G be a directed graph, and let r be a
                                       starting vertex
    agenda ← {r}                     ▷ Let agenda be a FIFO queue
    visited ← {r}
    while agenda is not empty do
        pop vertex s from agenda
        if s is a goal vertex then
            return s
        end if
        for each edge from s to t in G do
            if t ∉ visited then
                push t to agenda
                add t to visited
            end if
        end for
    end while
end function
```

Incidentally, if you change "agenda" to a LIFO stack, the above code implements depth-first search instead of breadth-first search.

You will need to apply the above pseudocode to NFA configurations, and modify it so that it can reconstruct the found path. If you change the "visited" variable 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-pa1.bash`. 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

If you use Python, please indicate which version of Python you tested your code with in the file `pa1/README.md` (you can print the version of Python by running `python -V` or `python3 -V`). If you use C++, provide a Makefile that automatically compiles your code with `g++`. The automatic tester will clone your repository, run `make -C pa1`, and then run `tests/test-pa1.bash`. 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 `pa1` (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 `pa1-1` for part 1, `pa1-2` for part 2, and so on.

Rubric

Part 1	
data structure	3
read_nfa	4
write_nfa	3
Part 2 (match)	
correct algorithm	7
handling ε	4
reconstructing path	7
Part 3 (`nfa_path`)	
correctness	4
time complexity	4
Total	36