Lecture 5: Closure under complement; Nondeterministic Automata

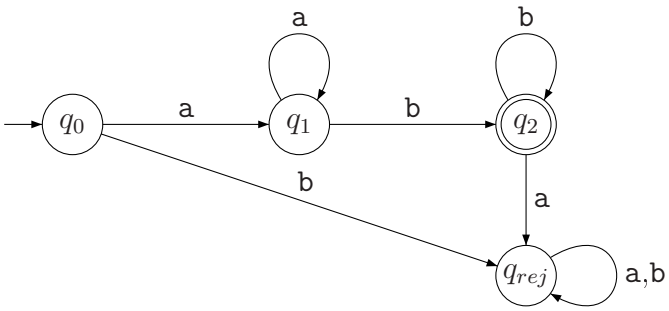
February 3, 2009

This lecture covers mainly the first part of section 1.2 of Sipser, through p 54.

1 Closure under complement of regular languages

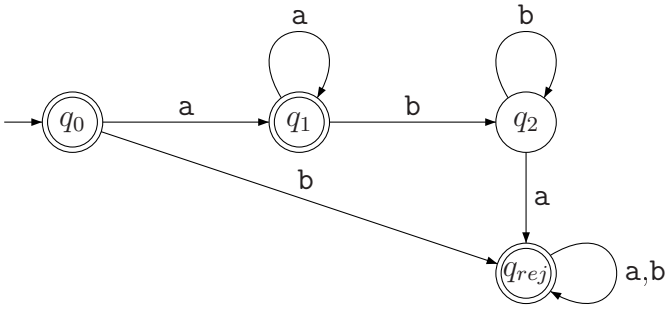
Here we are interested in the question of whether the regular languages are closed under set complement. (The complement language keeps the same alphabet.) That is, if we have a DFA $M = (Q, \Sigma, \delta, q_0, F)$ accepting some language L, can we construct a new DFA M' accepting $\overline{L} = \Sigma^* \setminus L$?

Consider the automata M from above, where L is the set of all strings of at least one a followed by at least one b.



The complement language $\overline{L}$ contains the empty string, strings in which some b's precede some a's, and strings that contain only a's or only b's.

Our new DFA M' should accept exactly those strings that M rejects. So we can make M' by swapping final/non-final markings on the states:



Formally, $M' = (Q, \Sigma, \delta, q_0, Q \setminus F)$.

Theorem 1.1 *Let $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q, \Sigma, \delta, q_0, Q \setminus F)$. Then $L(M') = \Sigma^* \setminus L(M)$.*

Proof: Note that since Q and δ are the same in M and M', M and M' have the same δ'. Also, note that they have the same initial state q_0.

Let $w \in \Sigma^*$.

Then $w \in L(M')$ iff $\delta^*(q_0, w) \in (Q \setminus F)$
iff $\neg(\delta^*(q_0, w) \in F))$
iff $\neg(w \in L(M))$
iff $w \in \Sigma^* \setminus L(M)$. ■

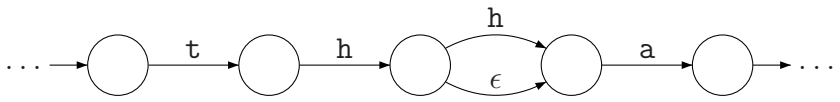
2 Non-deterministic finite automata (NFA)

A ***non-deterministic finite automata*** (***NFA***) is like a DFA but with three extra features. These features make them easier to construct, especially because they can be composed in a modular fashion. Furthermore, they are easier to read, and they tend to be much smaller and as such easier to describe. Computationally, they are equivalent to DFAs, in the sense that they recognize the same languages.

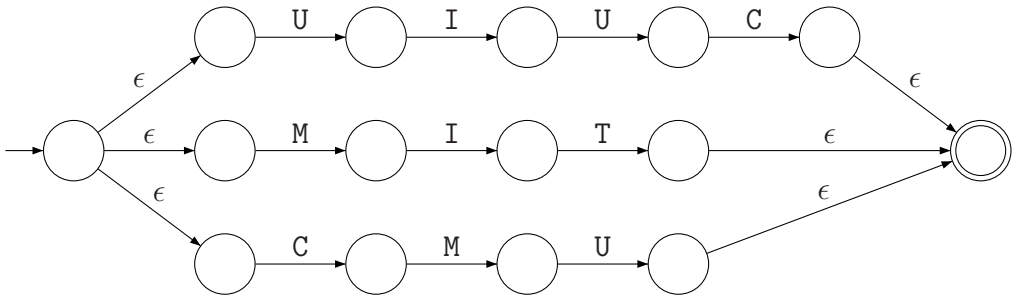
For practical applications of any complexity, users can write NFAs or regular expressions (trivial to convert to NFAs). A computer algorithm might compile these to DFAs, which can be executed/simulated quickly.

2.1 NFA feature #1: Epsilon transitions

An NFA can do a state transition without reading input. This makes it easy to represent optional characters. For example, "`Northampton`" is commonly misspelled as "`Northhampton`". A web search type application can recognize both variants using the pattern `North(h)ampton`.



Epsilon transitions also allow multiple alternatives (set union) to be spliced together in a nice way. E.g. we can recognize the set $\{\texttt{UIUC}, \texttt{MIT}, \texttt{CMU}\}$ with the following automaton. This allows modular construction of large state machines.



2.1.1 How do we execute an NFA?

Assume a NFA N is in state q, and the next input character is c. The NFA N may have multiple transitions it could take. That is, multiple possible next states. An NFA accepts if there is *some* path through its state diagram that consumes the whole input string and ends in an accept state.

Here are two possible ways to think about this:

(i) the NFA magically guesses the right path which will lead to an accept state.

(ii) the NFA searches all paths through the state diagram to find such a path.

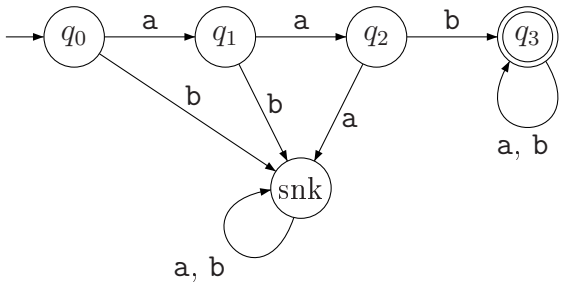
The first view is often the best for mathematical analysis. The second view is one reasonable approach to implementing NFAs.

2.2 NFA Feature #2: Missing transitions

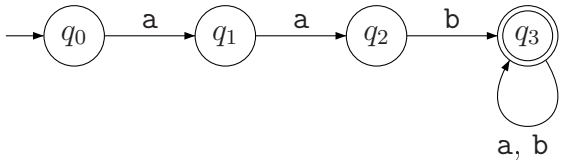
Assume a NFA N is in state q, and the next input character is c. The NFA may have no outgoing transition from q that corresponds to the input character c.

This means that you can not get to an accepting state from this point on. So the NFA will reject the input string unless there is some other alternative path through the state diagram. You can think of the missing transitions as going to an implicit sink state. Visually, diagrams of NFAs are much simpler by not having to put in the sink state explicitly.

Example. Consider the DFA that accepts all the strings over {a, b} that starts with aab. Here is the resulting DFA.

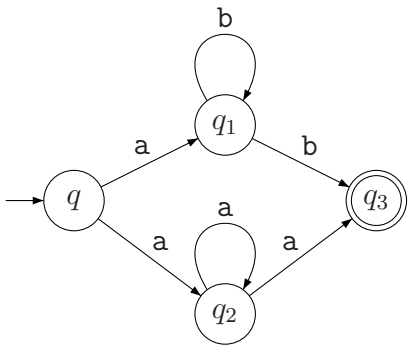


The NFA for the same language is even simpler if we omit transitions, and the sink state. In particular, the NFA for the above language is the following.



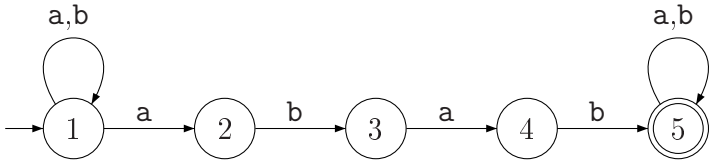
2.3 NFA Feature #3: Multiple transitions

A state q in a NFA may have more than one outgoing transition for some character t. This means that the NFA needs to "guess" which path will accept the input string. Or, alternatively, search all possible paths. This complicates deciding if a string is accepted by a NFA, but it greatly simplifies the resulting machines. Thus, the automata on the right accepts all strings of the form $\mathtt{a}b^i$ or $\mathtt{aa}^{\mathtt{i}}$ (for any $i \in \mathbb{N}$). Of course, its not too hard to build a DFA for this language, but even here the description of the NFA is simpler.

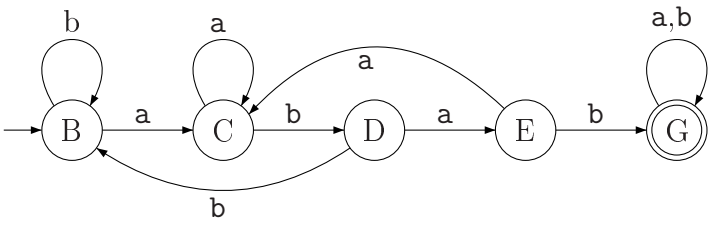


As another example, the automata below accepts strings containing the substring abab.

(N1)



The respective DFA, shown below, needs a lot more transitions and is somewhat harder to read.



3 More Examples

3.1 Running an NFA via search

Let us run an explicit search for the above NFA (N1) on the input string ababa. Initially, at time $t = 0$, the only possible state is the start state 1. The search is depicted in table on the right. When the input is exhausted, one of the possible states (E) is an accept state, and as such the NFA (N1) accepts the string ababa.

Time	Possible states	Remaining input
$t = 0$	$\{1\}$	ababa
$t = 1$	$\{1, 2\}$	baba
$t = 2$	$\{1, 3\}$	aba
$t = 3$	$\{1, 2, 4\}$	ba
$t = 4$	$\{1, 3, 5\}$	a
$t = 5$	$\{1, 2, 4, 5\}$	ϵ

3.2 Interesting guessing example

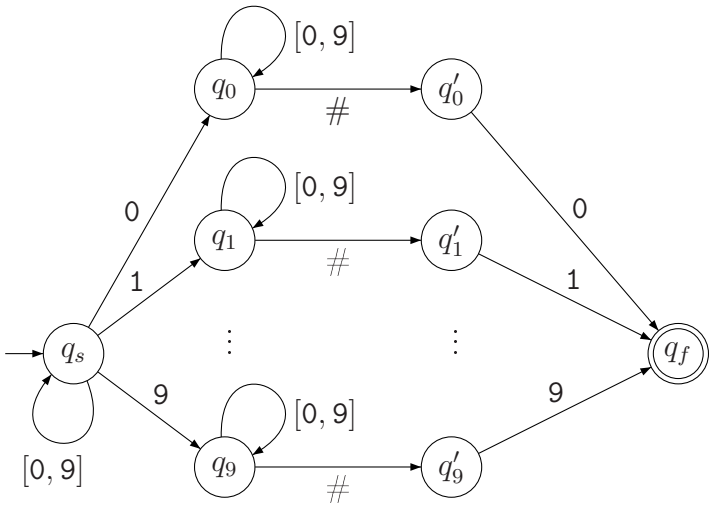
Some NFAs are easier to construct and analyze if you take the "guessing" view on how they work.

Let $\Sigma = \{0, 1, \ldots, 9\}$, denote this as $[0, 9]$ in short form. Let

$$L = \left\{ w\#c \;\middle|\; c \in \Sigma, w \in \Sigma^*, \text{ and } c \text{ occurs in } w \right\}.$$

For example, the word 314159#5 is in L, and so is 314159#3. But the word 314159#7 is not in L.

Here is the NFA M that recognizes this language.



The NFA M scans the input string until it "guesses" that it is at the character c in w that will be at the end of the input string. When it makes this guess, M transitions into a state q_c that "remembers" the value c. The rest of the transitions then confirm that the rest of the input string matches this guess.

A DFA for this problem is considerably more taxing. We will need a state to remember each digit encountered in the string read so far. Since there are 2^{10} different subsets, we will require an automata with at least 1024 states! The NFA above requires only 22 states, and is much easier to draw and understand.

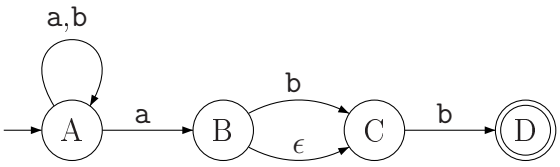
4 Formal definition of an NFA

An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. Similar to a DFA except that the type signature for δ is

$$\delta : Q \times \Sigma_\epsilon \to \mathbb{P}(Q),$$

where $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\mathbb{P}(Q)$ is the power set of Q (i.e., all possible subsets of Q). As such, the input character for $\delta(\cdot)$ can be either a real input character or ϵ (in this case the NFA does not eat [or drink] any input character when using this transition). The output value of δ is a *set* of states (unlike a DFA).

Example 4.1 Consider the following NFA:



Here

$\delta(A, \mathtt{a}) = \{A, B\}$

$\delta(B, \mathtt{a}) = \emptyset$ (NB: not $\{\emptyset\}$)

$\delta(B, \epsilon) = \{C\}$ (NB: not just C)

$\delta(B, \mathtt{b}) = \{C\}$ (NB: just follows one transition arc).

The trace for recognizing the input `abab`:

$t = 0$: state $= A$, remaining input `abab`.

$t = 1$: state $= A$, remaining input `bab`.

$t = 2$: state $= A$, remaining input `ab`.

$t = 3$: state $= B$, remaining input `b`.

$t = 4$: state $= C$, remaining input `b` (ϵ transition used, and no input eaten).

$t = 5$: state $= D$, remaining input ϵ.

Is every DFA an NFA? Technically, no (why?[1]). However, it is easy to convert any DFA into an NFA. If δ is the transition function of the DFA, then the corresponding transition of the NFA is going to be $\delta'(q, t) = \{\delta(q, t)\}$.

4.1 Formal definition of acceptance

Let $M = \big(Q, \Sigma, \delta, q_0, F\big)$ be an NFA. Let w be a string in Σ^*.

The NFA M accepts w if and only if there is a sequence of states $r_0, r_1, \ldots, r_n$ and a sequence of inputs $x_1, x_2, \ldots, x_n$, where each x_i is either a character from Σ or ϵ, such that

(i) $w = x_1 x_2 \ldots x_n$.

(The input string "eaten" by the NFA is the input string w.)

(ii) $r_0 = q_0$.

(The NFA starts from the start state.)

(iii) $r_n \in F$.

(The final state in the trance in an accepting state.)

(iv) $r_{i+1} \in \delta(r_i, x_{i+1})$ for every i in $[0, n-1]$.

(The transitions in the trace are all valid. That is, the state r_{i+1} is one of the possible states one can go from r_i, if the NFA consumes the character x_{i+1}.

So, in the above example, $n = 6$, our state sequence is $AAABCD$, and our sequence of inputs is abaϵb.

Key differences the notation of acceptance from DFA are

(i) Inserting/allowing ϵ into input character sequence.

(ii) Output of δ is a set, so in condition (iv) above, r_{i+1} is a member of δ's output. (For a DFA, in this case, we just had to check that the new state is equal to δ's output.)

[1] Because, the transition function is defined differently.