1 Introduction

Today we'll explore some more the relationship between deterministic and non-deterministic finite state automata, with and without λ-transitions. Remember that in the non-deterministic case, M accepts w if *some* computation of M on w leads to an accepting state. But first, an important definition.

2 Regular languages

The *regular languages* are a collection of languages formed by simple constructions from basic building blocks. We start with four basic languages (over $\Sigma = \{a, b\}$:

$$\emptyset, \boldsymbol{\lambda} = \{\lambda\}, \boldsymbol{a} = \{a\}, \boldsymbol{b} = \{b\}.$$

And three basic operations applied to languages L and K: union (i.e. $L \cup K$), concatenation (i.e. $L \cdot K$) and *Kleene star* denoted L^*. The definition of the latter is "concatenation of 0 or more elements of L" or formally by recursion: $\lambda \in L^*$, and if $w \in L^*$ and $u \in L$ then $wu \in L^*$.

A language is regular if it can be built up from basic languages using the three basic operations. Again formally by recursion:

Base The languages $\emptyset$, $\boldsymbol{\lambda}$ and $\boldsymbol{x} = \{x\}$ for $x \in \Sigma$ are regular languages

Construction If $\boldsymbol{u}$ and $\boldsymbol{v}$ are regular languages then so are $\boldsymbol{u} \cup \boldsymbol{v}$, $\boldsymbol{u} \cdot \boldsymbol{v}$ and $\boldsymbol{u}^*$.

Because it's convenient, we also define $L^+ = LL^*$, i.e. the language of one or more elements of L concatenated together.

Both START-a, the language of words whose first character is a, and ES, the language of words of even length, are regular languages:

$$\begin{aligned} \text{START-}a &= \boldsymbol{a}(\boldsymbol{a} \cup \boldsymbol{b})^* \\ ES &= ((\boldsymbol{a} \cup \boldsymbol{b})(\boldsymbol{a} \cup \boldsymbol{b}))^*. \end{aligned}$$

Two more examples: strings that contain bb as a consecutive substring, and strings that don't:

$$\begin{aligned} \text{HAS-}bb &= (\boldsymbol{a} \cup \boldsymbol{b})^* \boldsymbol{bb} (\boldsymbol{a} \cup \boldsymbol{b})^* \\ \text{HAS-NO-}bb &= (\boldsymbol{a} \cup \boldsymbol{ba})^* (\boldsymbol{b} \cup \boldsymbol{\lambda}). \end{aligned}$$

3 NFA-λ machines as modules

By introducing λ transitions we can "encapsulate" finite state automata in such a way that:

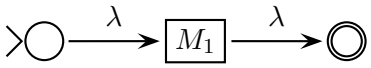
- there are no incoming transitions to the initial state
- there is a unique accepting state with no outgoing arrows

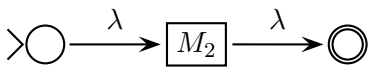
This is accomplished simply by adding a new initial state with a λ-transition to the previous initial state (and no other associated transitions), and adding a new accepting state, taking each original accepting state adding a λ-transition to it, and then making all of them no longer be accepting states. Clearly this new machine accepts exactly the same language as the original one. The advantage of enforcing the two conditions is that it makes it easy to combine NFAs.

Specifically we can prove:

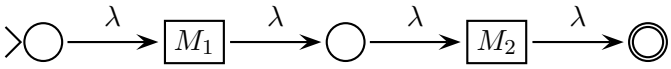
Theorem 3.1. *Suppose that M_1 and M_2 are NFAs accepting languages L_1 and L_2 respectively. Then there are NFAs that accept L_1L_2, $L_1 \cup L_2$ and L_1^*.*

Proof. Represent M_1 and M_2 as suggested above:

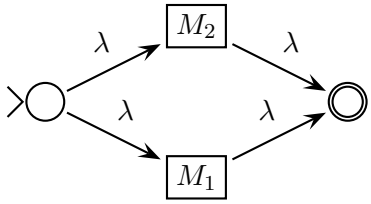




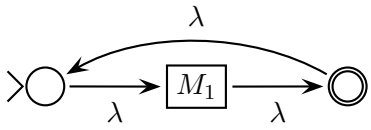
Now it is easy to combine these to recognize L_1L_2:



And $L_1 \cup L_2$:



And L_1^*:



□

Corollary 3.2. *Every regular language is accepted by some NFA.*

Proof. This is obvious since it's clear that there are NFAs which accept the "basic" regular languages $\emptyset$, $\boldsymbol{\lambda}$, and $\boldsymbol{x}$ for $x \in \Sigma$, and the previous result shows that the construction step in the recursive definition of regular languages can also be applied to NFAs. □

4 Eliminating non-determinism

Non-determinism is a bit sneaky – and it's hard to imagine how we could have "real" machines which implemented it. Fortunately it turns out that it's not actually required, as anything that can be recognized by an NFA-λ can actually be recognized by a DFA. So, we can freely make use of NFAs in proofs (because they are more flexible) secure in the knowledge that if necessary they can be implemented via DFAs.

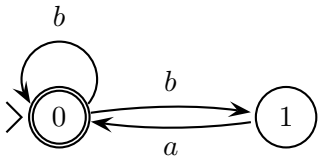
First let's see how to get rid of λ-transitions. Suppose that M is an NFA-λ and r is one of its states. Define the λ-closure of r to be the set of states that can be reached from r using only λ-transitions. Recursively: r is in the λ-closure of r, and if s is in theλ-closure of q, and $s \xrightarrow{\lambda} t$ then t is in the λ-closure of r.

Now define a new automaton M' that has the same set of states as M (and the same initial and accepting states) but without λ-transitions as follows. For each state q of M and each letter a in Σ define:

$$\delta'(q, a) = \{t \mid \text{for some } r \text{ in the } \lambda\text{-closure of } q,\ t \text{ is in the } \lambda\text{-closure of some } s \in \delta(r, a)\}$$

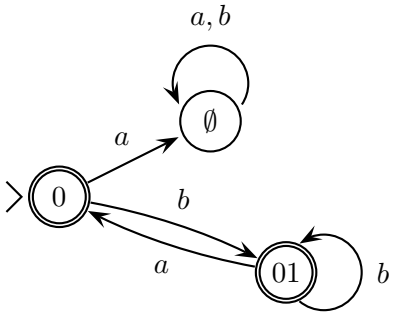
i.e. $\delta'(q, a)$ is just the set of all states we could get to from q by following some λ-transitions, then an a-transition, and then some more λ-transitions. It's clear that M and M' accept the same language, and M' has no λ-transitions.

How can we get from an NFA to a DFA? The idea is to trace all possible paths of a computation simultaneously, using states of the DFA to represent sets of states in the NFA – all the "places we might be" at this point. Before doing this formally here's a simple example.



It's easy to see that this automaton accepts $(\boldsymbol{b} \cup \boldsymbol{ba})^*$ and it's non-deterministic both because there are two outgoing b-transitions from state 0 and no a-transition there.

Now let's examine what sets of states we might be in after (partially) processing a word. Certainly we could be in state 0 alone (e.g. when we begin). From that state if we get an a we're nowhere, so we could be in the empty set of states (denote this by $\emptyset$). If we get a b we could be in state 0 or state 1 (denote by 01). We don't need to worry about what happens from the empty set (if we get either a or b we stay there – this is a "cannot accept" state). From the state 01, a b could leave us in 0 (if that's where we were) or take us to 1 (again if we were in 0), and a could take us to 0 (if we were in 1). So, we get to 01 on b and to 0 on a. Now there are no remaining states to worry about so let's try to draw the new automaton:



There was one remaining subtlety – each state of this automaton that includes an accepting state must be made into an accepting state.

The idea behind this construction gives us:

Theorem 4.1. *Let M be an NFA. Then there is a DFA DM that accepts the same language.*

Proof. Rather than explicitly working out "where we might be", in proving this result it's easiest to assume "we might be anywhere". That is, if Q is the set of states of M we take the set of states of DM to be $\mathcal{P}(Q)$, i.e. the set of all subsets of Q. Then, for each $c \in \Sigma$ and $X \subseteq Q$ we define:

$$\delta_{DM}(X, c) = \{y \in Q \mid \text{for some } x \in X,\, y \in \delta_M(x, c)\}$$

Taking the initial state of DM to be $\{q_0\}$ where q_0 is the initial state of M, and the set of final states of DM to be the set of all $X \subseteq Q$ such that X contains some final state of M we have completed the construction.

By construction, each transition from a state X of DM identifies "where we might be" if we started in one of the states of M that belongs to X and processed the given character. So, an accepting computation in DM represents some accepting computation in M and vice versa. □

It's worth noting that the number of states we use in DM is 2^m where m is the number of states in M. It can be shown that this exponential blow up is *necessary* in general.

For a fixed positive integer N, consider the following language, L, over $\{a, b\}$. A word $w \in L$ if there is some pair of b's in the word so that the total number of a's between them is a multiple of N (there may be other b's between them as well - and there need not be any a's, i.e. bb is in the language). It's not immediately obvious that this language is regular, but it's easy to design an NFA that accepts it. This NFA has $N + 2$ states called S, 0, 1, 2, $\ldots N - 1$, and F.

The start state S, has a, and b loops, as well as a b arrow to state 0. The states 0, 1, 2, through $N - 1$ form a cycle with a arrows from each to the next (and from $N - 1$ to 0), and b loops. Finally there is a b arrow from 0 to F, and a and b loops on F. State F is the only accepting state.

To accept a word by this automaton we need to hang around S for a while, kick over to the cycle using a b, run around the cycle some number of times (thereby consuming a multiple of N a's, while ignoring b's) and finally kick off the cycle using another b. In other words, we accept L.

Now consider a DFA that accepts L. Since it is not allowed to "guess", it must know at any point what the possible lengths of "a's following a prior b" are modulo N. Given

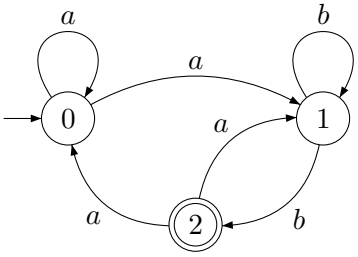
any subset $X \subseteq \{0, 1, 2, \ldots, N-1\}$ it's easy to construct a word w_X where the set of such lengths is X. Just for example, if $N = 12$ and $X = \{1, 4, 6, 7\}$ we can take the $w_X = babaabaaaba$, which has 7 a's after the first b, 6 after the second etc.

Now, I claim that if $X \neq Y$ then the state we are in when we have processed w_X must be different from that we are in when we process w_Y. The reason is that there is some word v such that $w_X v \in L$ and $w_Y v \notin L$ or vice versa (if we were in the same state we would have to accept or reject both of these words). For instance if k belongs to X but not to Y (or vice versa) we can take $v = a^{N-k}b$.

So, any DFA to accept L must have at least 2^N states, and an exponential blow up cannot be ruled out.

5 Tutorial problems

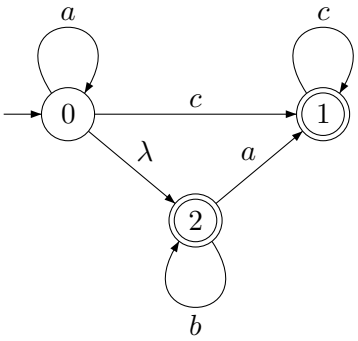
1. Let M be the nondeterministic finite automaton:



   - Trace all computations of the string $aaabb$ in M. Is it in $L(M)$?
   - Give a regular expression for $L(M)$.

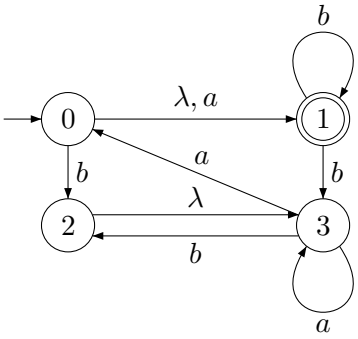
2. Design an NFA that accepts the language of strings over $\{a, b, c\}$ whose length is a multiple of three, and which can be divided into blocks of length three, each of which contains each symbol exactly once.

3. Let M be the NFA



   - Compute the λ-closure of each state.
   - Construct a DFA that is equivalent to M
   - Give a regular expression for $L(M)$.

4. Repeat the preceding question with the following automaton:



5. Build an NFA that accepts $(ab)^*$ and one that accepts $(ba)^*$. Use λ-transitions to combine them into an NFA accepting $(ab)^*(ba)^*$. Convert that NFA to an equivalent DFA.

6. It's very easy to build an NFA that accepts the language HAS-$abba$ (just loop on a or b in the initial state, then a chain of transitions to accept specifically $abba$, and then loop on a or b in the final state). Convert this NFA to a DFA and compare the result to the example constructed in Lecture 5.