More on automata — Regular Expressions, limitations on automata

March 24 — April 7

1 Regular expressions

One of the reasons that automata are an interesting computational model is that we can precisely characterize the languages that we can build machines to recognize. These languages are defined by *regular expressions*.

A regular expression r is a pattern that matches a set of strings. For example, the regular expression $ab(a^*)$ matches the strings ab, and aba, and $abaaa$, and any string that starts with ab followed by any number of as. The $*$ (called "Kleene star") indicates that the pattern should be repeated 0 or more times. The set of strings matched by an expression r is called the language of the expression and is denoted $L(r)$.

In general, a regular expression is any of the following:

- a single character $a \in \Sigma$. The language of the expression a is just $\{``a''\}$.
- the concatenation of two regular expressions $r_1 r_2$, which matches any string formed by concatenating a string in $L(r_1)$ with a string in $L(r_2)$. Formally, $L(r_1 r_2) = \{xy \mid x \in L(r_1) \text{ and } y \in L(r_2)\}$.
- r'^* where r' is any regular expression. As described above, $L(r'^*) = \{x_1 x_2 x_3 \cdots x_n \mid x_i \in L(r')\}$.
- $r_1 + r_2$ where r_1 and r_2 are regular expressions. This is called the union (or sometimes the alternation) of r_1 and r_2. It matches any string matched by either r_1 or r_2: $L(r_1 + r_2) = L(r_1) \cup L(r_2)$.
- The expression $\emptyset$ which matches no strings.
- The expression ϵ which matches only the empty string.

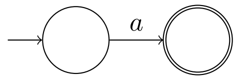
A language L is called *regular* if there is some regular expression r with $L(r) = L$.

1.1 Regular expression to NFA conversion

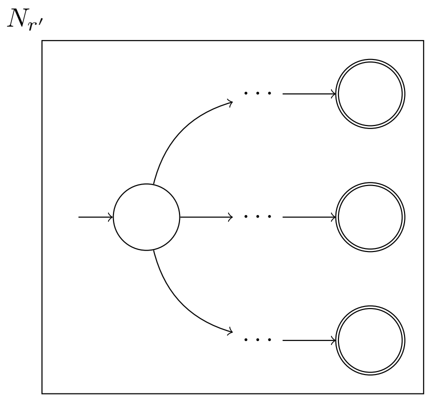
We want to show that the set of NFA-recognizable languages is the same as the set of regular languages. To do this, we must show that every regular language is recognizable, and that every recognizable language is regular.

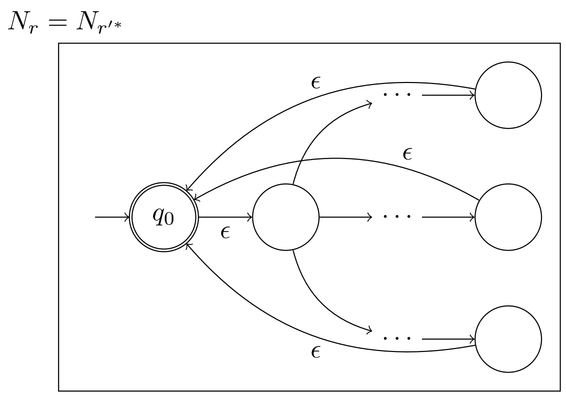
To show that every regular language L is recognizable, we will construct a machine to recognize it. Since L is regular, there is an expression r that matches it. We'll consider the cases for r:

- If $r = a$, then the following NFA has the same language as r:



- You will do the case $r = r_1 r_2$ in the homework
- If $r = r'^*$ then we will inductively assume that we have a machine N' with $L(N') = L(r')$. To construct N from N', we will add a new start state q_0 to N'. We will make q_0 into an accepting state, and add an epsilon transition from q_0 to the start state of N'. We will add transitions from each of N''s accepting states to q_0.





- If $r = r_1 + r_2$, we inductively assume that we have machines N_1 and N_2 with languages $L(r_1)$ and $L(r_2)$ respectively. We can then use the union construction above to construct N with $L(N) = L(r_1) \cup L(r_2) = L(r_1 + r_2)$.
- If $r = \emptyset$ then a single non-accepting state with no transitions recognizes $L(r)$.
- If $r = \epsilon$ then we can use the example machine in the DFA minimization discussion above.

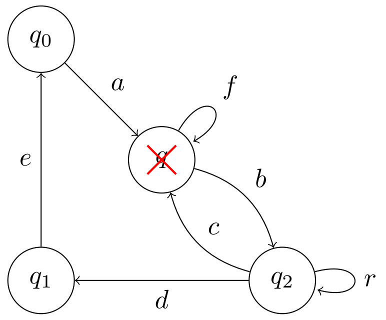
In each case, we have constructed a machine N with $L(N) = L(r)$. Thus every regular language is NFA-recognizable.

1.2 NFA to regular expression conversion

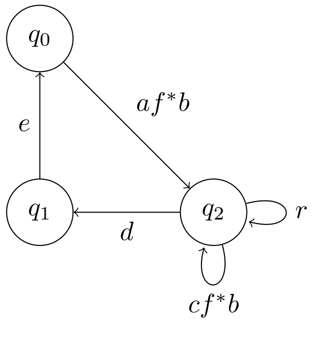
To show that every NFA-recognizable langauge is regular, we must build a regular expression out of an NFA.

In order to do so, we'll build a "generalized NFA" whose edges are labelled by regular expressions instead of just symbols from Σ. We will start with the given NFA and repeatedly remove states until only the start state and an accepting state remain. We will then be able to read the regular expression from the single transition from the start state to the accepting state.

At each step, we will keep the language of the machine the same. To do so, we must add in new transitions that capture the paths that went through the removed state. For example, suppose we wanted to remove the boxed state q from the following machine:

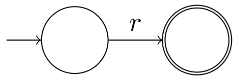


To do so, we must add in edges capturing all of the paths from the remaining nodes, through q, and back to the remaining nodes that are lost. For example, it is possible to transition from q_0 to q_2 by passing through q using any string matching af^*b. So we will add a new transition from q_0 to q_2 annotated with this regular expression. Similarly, we can transition from q_2 back to itself with the string cf^*b, so we will add a self loop to q_2 with the annotation cf^*b. In general, we must consider every pair of remaining states q', q'', and add a new edge corresponding to any path from q' to q, through a loop on q, and back to q'. In this case, we have the following:



Note that there are now two edges from q_2 to itself. This means that we can transition from q_2 using either a string matching r *or* a string matching cf^*b. This is equivalent to a single transition labelled by $r + cf^*b$. In general we can combine all duplicate transitions this way.

We want to remove states until we are left with a single start and accept state. To avoid having to remove accepting states we will add a single new accepting state with an epsilon transition from each old accepting state into it. We will also add a new start state with an epsilon transition to the old start state; this guarantees that there are no transitions back into the start state. These two modifications allow us to remove states until we reach the following machine:



This machine clearly accepts $L(r)$. It also has the same language as our starting machine, so $L(r) = L(N)$, which was our goal.

Thus every NFA-recognizable language is regular.

2 Limits of automata

We know that DFA are "powerful" in the sense that they can recognize any regular language. Can we give an example of what they can't do?

Consider the language $L = \{x \in \{a,b\}^* \mid \#a(x) = \#b(x)\}$ (here $\#a(x)$ denotes the number of times a appears in x). We can try to build a machine to recognize L, but it seems like we always need to add states.

In fact, we can prove there is no such machine. Proof by contradiction. Assume there is a machine M. Since Q_M is finite, there are only n states. The string $x = a^{n+1}b^{n+1}$ should be accepted.

While processing the $n+1$ as, the machine must hit the same state q twice. That means that we can split up x:

$$x = \underbrace{aaa\cdots}_{w}\underbrace{\cdots}_{y}\underbrace{\cdots abbb\cdots b}_{z}$$

so that the processing of y starts and ends in q.

This means that the string wz is also accepted by M, but $wz \notin L$. So we have a contradiction.

This is an example of a general technique called the "pumping lemma".