CS 4510 Automata and Complexity	January 19
Lecture 3	
Lecturer: Frederic Faulkner	*Scribe(s):*

1 Regular operations

Last class we saw that regular languages are closed under complement. *Closed* here means that if we start with a regular language, and apply some operation to it (in this case, the complement operation), then the resulting language is guaranteed to still be regular. Today we will discuss several more operations that are preserve regularity, namely *union*, *concatenation*, and *Kleene star*. Then we will introduce a new model of computing, *nondeterministic finite automata*, that will help us to prove that regular languages are in fact closed under these operations.

The union of two languages L_1 and L_2, denoted $L_1 \cup L_2$, is defined to be the set $\{x | x \in L_1$ or $x \in L_2\}$, that is, $L_1 \cup L_2$ contains any string that is in either or both of L_1 or L_2. As an example, if $L_1 = \{cat, dog\}$ and $L_2 = \{fish, bird\}$, then $L_1 \cup L_2 = \{cat, dog, fish, bird\}$.

The concatenation of two languages L_1 and L_2, denoted $L_1 \circ L_2$ (or just L_1L_2), is the set $\{xy | x \in L_1$ and $y \in L_2\}$. For example, if $L_1 = \{cat, dog\}$ and $L_2 = \{fish, bird\}$, then $L_1 \circ L_2 = \{catfish, dogfish, catbird, dogbird\}$. Note that order matters: $fishdog$ is an element of $L_2 \circ L_1$ but not $L_1 \circ L_2$.

Finally, the Kleene star of a language L, denoted L^*, is the set $\{\varepsilon\} \cup L \cup LL \cup LLL \cup ...$, i.e. the set of all concatenated sequences of 0 or more elements of L (repeats allowed). Thus, if $L = \{cat, dog\}$, then $L^* = \{\varepsilon, cat, dog, catcat, catdog, dogcat, dogdog, catcatcat, catdogcat, ...\}$. **Self-quiz:** In general, L^* is an infinite set for any language L. Is this always the case? What is the smallest possible size of L^*? (Hint: not zero.)

Theorem 1 *Regular languages are closed under union, concatenation, and Kleene star. I.e. suppose L_1 and L_2 are regular languages. Then*

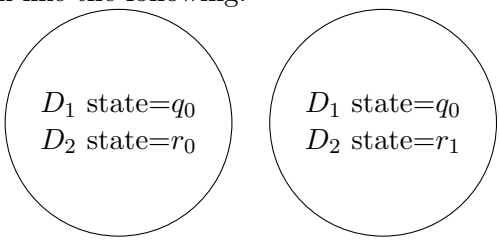
1. *$L_1 \cup L_2$ is regular*
2. *$L_1 \circ L_2$ is regular*
3. *L_1^* is regular.*

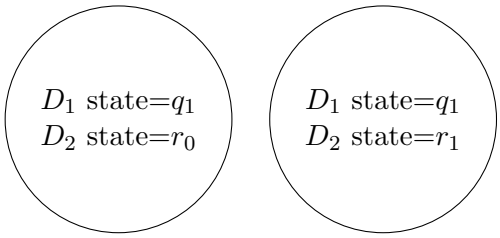
Using what we have studied so far in class, we can prove theorem 1 for union, so we will do that now. However, we will need additional tools to prove concatenation and Kleene star.

1.1 Proving that union preserves regularity

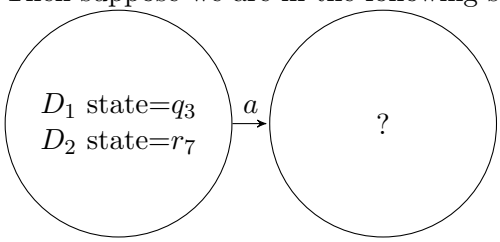
Idea: Let L_1 and L_2 be regular languages with DFA's D_1 and D_2 respectively. We wish to make a DFA D for $L_1 \cup L_2$. For any input x, when should D accept x? Well, it should accept x if either D_1 or D_2 accepts x. It would be nice if we could just run D_1 on x, then go back to the beginning and simulate D_2 on x, then check if we accepted either time. Unfortunately, we can't do that because we only are allowed to read the string once. In some sense, we must perform the computations of D_1 and D_2 *simultaneously*. Can we do that? We can!

As we read x, in each state of D we will just keep track of both the state that D_1 would be in at this point of x and the state that D_2 would be in. I.e. we will have states that look like the following:

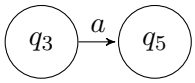


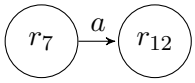


Then suppose we are in the following state of D and we read an a:

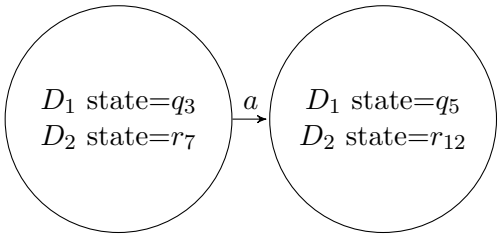


But we know that the original machines D_1 and D_2 have the following transitions:





Then we can the following transition to D:



So we have now decided how the states, alphabet, and transitions of D are formed; what remains of the proof? The start and final states, of course! The start state will just be the pair of start states of the original machines. What about accepting states? We want to accept x whenever at least one of D_1 or D_2 accepts. So we should accept whenever either (or both) of our "internal" states is an accepting state.

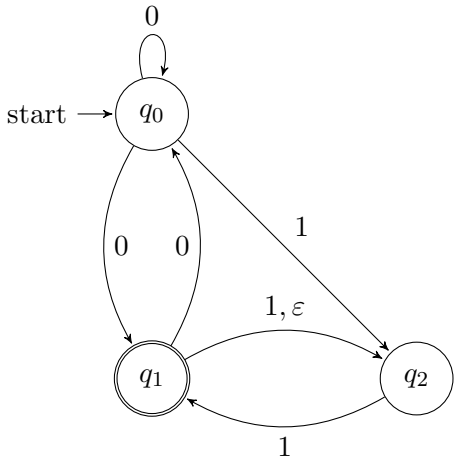
Formal Proof. Let L_1 and L_2 be regular languages with corresponding DFA's $D_1 = (Q, \Sigma, q_0, F, \delta)$ and $D_2 = (R, \Sigma, r_0, G, \gamma)$ respectively. Then we will construct a DFA D for $L_1 \cup L_2$ as $(Q \times R, \Sigma, (q_0, r_0), F', \delta')$, where $F' = \{(q, r) | q \in F \text{ or } r \in G\}$ and $\delta'((q, r), a) = (\delta(q, a), \gamma(r, a))$.

2 Nondeterminism

We now discuss a new model of computing: *nondeterministic finite automata* (NFA). As the name suggests, they are closely related to DFA's, but they have some expanded capabilities. In particular,

- It is not required to have a transition for every input at each state. There can be "missing" transitions.
- In a given state, there might be multiple transitions labeled with the same input.
- There is now a new type of transition called an ε-transition, which is optional and does not consume the next character of the input.

NFA's are basically DFA's that may have options available to them at each step of the computation and can make choices. For example, consider the following NFA:



There are multiple state sequences it can enter on the input 001:

- $q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_2$ (by looping twice)
- $q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{1} q_1$ (by looping once, use optional ε-transition)
- $q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2$ (by looping once, skip optional transition)
- $q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_0 \xrightarrow{1} q_2$ (no loops)

So which sequence does the NFA use? Well, all of them, in a sense. An NFA accepts a string if there is *any* sequence of choices it can make that causes it to end in an accept state. It only rejects a string if *every* possible sequence of choices ends in a non-accepting state or gets "stranded" (no available transition for the next input).

Thus, the NFA above accepts 001, because the second sequence of choices ends in an accept state.

On the other hand, consider the possible state sequences for the input 010:

- $q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_2$ (machine stranded, no 0-transition in q_2)
- $q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2$ (machine stranded, no 0-transition in q_2)
- $q_0 \xrightarrow{0} q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{1} q_1 \xrightarrow{0} q_0$ (end in non-accept state)

So 010 is rejected by the machine.

Definition 2 *Formally, a nondeterministic finite automata N is a 5-tuple $(Q, \Sigma, q_0, F, \delta)$ where*

- *Q is the set of states*
- *Σ is the input alphabet*
- *$q_0 \in Q$ is the start state*
- *$F \subseteq Q$ is the set of final states*
- *$\delta : Q \times \Sigma \to P(Q)$ is the transition function*

You may notice that this is almost identical to the definition of a DFA. The only real difference is that the transition function now has a co-domain of $P(Q)$ instead of Q. All this means is that, unlike in a DFA where there is only one "next" state for any given state and input, in an NFA there may be multiple states (or none) which are reachable from any given state and input.

Definition 3 *An NFA N is said to accept x, if there exists a sequence $q_0, ..., q_{|x|}$ of states of N such that q_0 is the initial state of N, $q_{|x|}$ is an accepting state of N, and $q_{i+1} \in \delta(q_i, x[i])$ for $i \in [0...|x| - 1]$ where δ is the transition function of N.*