CSE491/596 Lecture Wed. 9/16/20 Regular and Nonregular Languages

[cover FA-to-regexp example at end of previous lecture notes]
[Generally, things in square brackets are either skims of things I did in the lecture or extra comments afterward.]

Given a DFA $M = (Q, \Sigma, \delta, s, F)$, let us use the notation $\delta^*(p, x) =$ the state q that M is in after processing x from state p. We saw this as Δ^* for the DFA in the NFA-to-DFA proof. Note that

$$x \in L \iff \delta^*(s, x) \in F,$$

where $L = L(M)$, so

$$x \notin L \iff \delta^*(s, x) \notin F,$$

which is the same as writing

$$x \in \widetilde{L} \iff \delta^*(s, x) \in \widetilde{F}.$$

The upshot is that the DFA $M' = \left(Q, \Sigma, \delta, s, \widetilde{F}\right)$ gives $L(M') = \widetilde{L}$. This trick of complementing accepting and nonaccepting states does not, however, work for a general NFA. For example, if you try this on the NFAs N_k given for the languages L_k of binary strings whose kth bit from the end is a 1, then the new machine has an accepting loop at the start state on both 0 and 1 and so accepts every string, not just those in the complement of L_k. [I spent some time showing this from the picture of N_k in the previous lecture.] But thanks to Kleene's Theorem, being able to do it for DFAs is enough to prove:

Theorem 1: The complement of a regular language is always regular. ☒

Now suppose we have two DFAs $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ (note that Σ is the same). Let $L_1 = L(M_1)$ and $L_2 = L(M_2)$. Then let ***op*** be any binary operation on sets, such as $\cup$ or $\cap$ but note also difference $L_1 \setminus L_2$ and *symmetric difference*

$$L_1 \triangle L_2 = (L_1 \setminus L_2) \cup (L_2 \setminus L_1) = (L_1 \cup L_2) \setminus (L_1 \cap L_2),$$

whose corresponding Boolean operation ***op'*** is XOR, which is sometimes written $\oplus$. Then we have:

$$x \in L_1 \,\mathbf{op}\, L_2 \iff (x \in L_1 \,\mathbf{op'}\, x \in L_2) \iff \left(\delta_1^*(s_1, x) \in F_1\right) \mathbf{op'} \left(\delta_2^*(s_2, x) \in F_2\right)$$

When ***op'*** = AND, this is $\iff (\delta_1^*(s_1, x), \delta_2^*(s_2, x)) \in F_1 \times F_2$.
This means that **if** we define

$M_3 = (Q_3, \Sigma, \delta_3, s_3, F_3)$ with $Q_3 = Q_1 \times Q_2$ and $s_3 = (s_1, s_2)$,

and define $\delta_3((q_1, q_2), c) = (\delta_1(q, c), \delta_2(q, c))$,

and use $F_3 = F_1 \times F_2$,

then $L(M_3) = L(M_1) \cap L(M_2)$.

We can use this **Cartesian product construction** for the other Boolean operations op'. We just have to be more careful about how we define the final states. The general definition is

$$F_3 = \{(r_1, r_2) : r_1 \in F_1 \; op' \; r_2 \in F_2\}.$$

Then $L(M_3) = L(M_1) \; op \; L(M_2)$. Thus we have shown the following theorem.

Theorem 2: The class of regular languages is closed under all Boolean operations.

Actually, we already could have said this right after Theorem 1 about complements. This is because OR is a native regular expression operation. OR and negation ($\neg$) form a complete set of logic operations. For instance, a AND $b \equiv \neg((\neg a)$ OR $(\neg\, b))$ by DeMorgan's laws.

Summary of the philosophical discussion that ended the lecture: Suppose L_1 and L_2 are the two regular languages you want to combine. If what you're given are DFAs M_1 and M_2 for them ("for them" means $L(M_1) = L_1$ and $L(M_2) = L_2$), then the combination M_3 can be quickly put together as above, and it doesn't matter what the operation is. But if you are originally given NFAs N_1 and N_2, it is not so easy. Well, it is easy for union/OR if you only need an NFA N_3: just join N_1 and N_2 in parallel with ϵ-arcs from a new start state as we saw in the NFA-to-regexp proof. For intersection/AND, hmmm....[Is there a way to make the Cartesian product construction idea work directly on two NFAs? That might be a good small-group discussion topic.] And if the operation is difference or symmetric difference---or just simply complement like on the homework problem 3---there seems to be no way *in general* without first converting the NFAs to DFAs so you can apply the Cartesian product idea. Maybe in particular cases there are shortcuts, but in this course we emphasize geenral cases.

This all highlights a curious asymmetry between OR and AND. The former is a native regular expression operation. We at least a philosophical analogy to parallel circuits in Kirchoff's Laws. There does not, however, seem to be an electrical counterpart to AND. Even if you can do Cartesian product on two NFAs to handle AND, the new NFA is quadratically bigger than the two given NFAs. This may be more than analogy---it may be responsible for differences in cases where our brains find disjunctions easier to think about than conjunctions. [The difference actually really comes out when we contrast *disjunctiive normal form,* which means OR-of-ANDs, with *conjunctive normal form*, which is AND-of-ORs. We will hit this when doing NP-completeness.]