Lecture 4

Equivalence of NFAs, DFAs, and Regular Expressions

In this lecture, we will show that NFAs, DFAs, and regular expressions are equivalent in their expressive power: that is, a language A can be recognized by a DFA if and only if it can be recognized by and NFA, and also if and only if it has a regular expression. We will start by showing the equivalence of NFAs and DFAs.

4.1 NFAs and DFAs

Each DFA can be easily converted to an NFA recognizing the same language (indeed, the state diagram does not need to change at all, only the formal definition of δ needs to change slightly). In order to show that NFAs and DFAs are equivalent in power, we will only need to show that every NFA can be converted into a DFA recognizing the same language. That is, let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA; our goal will be to construct a DFA $M = (Q', \Sigma', \delta', q_0', F')$ such that $L(M) = L(N)$.

Since we want the DFA M to recognize the same language as N, we will clearly want to set $\Sigma' = \Sigma$ (that is, we want the alphabet of M to be the same as that of N). So what we need to do is to describe a set of states Q', a transition function δ', a set of accept states F', and a start state q_0' such that together they characterize, in a deterministic way, the nondeterministic behavior of N.

How shall we do this? The trick is to think about how you would describe the status of N when it is in the middle of a run. What is the "memory" that N keeps when it is in the middle of reading a string? Once we think about it this way, the answer should be clear: what N remembers when it is in the middle of a run is *the set of all possible states it might be in.*

Of course, a DFA can only remember which state it is in, and nothing else. But this suggests one way of simulating an NFA using a DFA: we will simply create a DFA that has one state for each *set* of states the NFA has. That is, the set Q' of states of the DFA will be $\mathcal{P}(Q)$, the set of all subsets of Q, where Q is the set of states of the NFA. This means that $|Q'| = 2^{|Q|}$, so the DFA will have exponentially more states than the NFA.

To make sure that this DFA M simulates the NFA N, we will ensure that at each point in time—after reading any fixed string x—the state of the DFA will correspond to the set of states the NFA is allowed to reach after reading x, that is, the set $\delta^*(q_0, x)$. To ensure this, we will first set the start set of the DFA to be $q_0' = \epsilon(\{q_0\})$, the ϵ-closure of the start state of the NFA, which is the set of all states the NFA can reach without reading any symbols. Next, we will define the transition function δ' of the DFA. This will be defined by $\delta'(S, c) = \epsilon\left(\bigcup_{q \in S} \delta(q, c)\right)$. To understand this definition, note that each $S \in Q'$ is also a subset $S \subseteq Q$; this state S of the DFA gets mapped to a different

state T, which is also a subset of states of the NFA, $T \subseteq Q$. This T is the set of all states of the NFA that can be reached from a state in S by reading c, and then by any number of ϵ-transitions. Finally, we define F', the set of accept states of the DFA M. This will be the set of all sets of states of the NFA contain at least one accept state of the NFA; that is, $F' = \{S \subseteq Q : S \cap F \neq \varnothing\}$.

Having fully defined the DFA M in terms of N, we can now argue that the accept exactly the same strings; that is, $L(M) = L(N)$. To show this, we will first show that the behavior of the two machines is effectively the same. Formally, we claim that for any string $x \in \Sigma^*$, we have $(\delta')^*(q_0', x) = \delta^*(q_0, x)$. This is saying that for any string x, the state reached by M when run on x is exactly the set $\delta^*(q_0, x)$ of states of N reachable by N when run on x.

We do this by induction on the length of the string x. In the base case, when $x = \epsilon$, we have

$$(\delta')^*(q_0', \epsilon) = q_0' = \epsilon(\{q_0\}) = \delta^*(q_0, \epsilon),$$

as desired. When $|x| > 0$, we can write $x = yc$ for $y \in \Sigma^*$ and $c \in \Sigma$. Then by the induction hypothesis, we can assume that $(\delta')^*(q_0', y) = \delta^*(q_0, y)$. Now, expanding the definition of the extended transition function of a DFA, we get

$$(\delta')^*(q_0', yc) = \delta'((\delta')^*(q_0', y), c) = \delta'(\delta^*(q_0, y), c),$$

where we used the induction hypothesis in the second equality. Next, using the definition of δ', we have

$$\delta'(\delta^*(q_0, y), c) = \epsilon\left(\bigcup_{q \in \delta^*(q_0, y)} \delta(q, c)\right),$$

which by the definition of the extended transition function δ^* of an NFA, is equal to $\delta^*(q_0, yc)$. Hence we have shown $(\delta')^*(q_0', yc) = \delta^*(q_0, yc)$, so by induction, we conclude that $(\delta')^*(q_0', x) = \delta^*(q_0, x)$ for all strings $x \in \Sigma^*$.

From here, showing that $L(M) = L(N)$ is easy. By definition, $x \in L(M)$ if and only if $(\delta')^*(q_0', x) \in F'$. Since $(\delta')^*(q_0', x) = \delta^*(q_0, x)$, we have $x \in L(M)$ if and only if $\delta^*(q_0, x) \in F'$. By the definition of F', this happens if and only if $\delta^*(q_0, x) \cap F \neq \varnothing$. Finally, by the definition of $L(N)$ for an NFA, we have $x \in L(N)$ if and only if $\delta^*(q_0, x) \cap F \neq \varnothing$, completing the chain.

Example. To get a clearer understanding of how this conversion from an NFA to a DFA works, let's apply it to a small example. Consider the NFA in Figure 4.1.

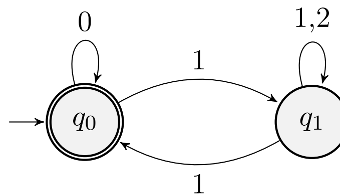

Figure 4.1: An NFA over the alphabet $\{0, 1, 2\}$.

How can we turn this NFA into a DFA? Using the construction described above, we will have one state in the DFA for each *set* of states of the NFA. This means the states of the DFA will be $\varnothing$, $\{q_0\}$, $\{q_1\}$, and $\{q_0, q_1\}$. Of these, the sets containing q_0 will be accepting, and the start state will be $\{q_0\}$ (since there are no ϵ-transitions coming out of q_0).

The state diagram of the DFA is given in Figure 4.2. The transitions in it were computed by thinking, for each set of states of the NFA and each alphabet symbol, where can the NFA go from

that set of states when seeing that alphabet symbol? Then we draw an arrow to set of states the NFA can go.

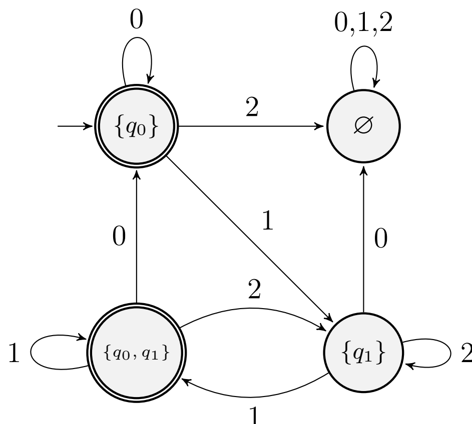

Figure 4.2: A DFA recognizing the same language as the NFA in Figure 4.1.

4.2 From regular expressions to NFAs

In the previous section, we saw that DFAs and NFAs have same power for recognizing languages: every language that can be recognized by one can be recognized the other. Next, we will show that every language that has a regular expression can be represented by an NFA; this will show NFAs have at least the same representational power as regular expressions. In the next section, we will show the converse, that every NFA can be converted into a regular expression as well.

Which languages have a regular expression? By definition, these are the regular languages: languages that can be formed using the operations union, concatenation, and star, starting from only the languages $\varnothing$, $\{\epsilon\}$ and $\{c\}$ for each $c \in \Sigma$. We now want to show that all such regular languages can be recognized by NFAs.

We will start with the "base case" languages: $\varnothing$, $\{\epsilon\}$ and $\{c\}$ for each $c \in \Sigma$. Clearly, there is an NFA for $\varnothing$: in fact, any NFA with no accept states recognizes $\varnothing$. This is because with no accept states, the NFA will reject all strings. Next, consider the language $\{\epsilon\}$. This language accepts the empty string and nothing more; it can be recognized by the NFA in Figure 4.3.

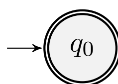

Figure 4.3: An NFA recognizing $\{\epsilon\}$.

Note that since there are no arcs in this NFA, seeing any symbol causes it to drop into the abyss, rejecting forever. Since this NFA might be a little confusing, let's also write a DFA for the language $\{\epsilon\}$. See Figure 4.4.

Next, consider the language $\{c\}$ for some $c \in \Sigma$. This can be recognized by the NFA in Figure 4.5.

Now that we've handled the base case languages, let's consider the operations union, concatenation, and star. Suppose we have two languages A and B over an alphabet Σ, and suppose that we already have NFAs for our languages, denoted N_A and N_B. How might we construct an NFA

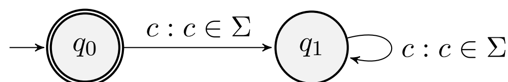

Figure 4.4: A DFA recognizing $\{\epsilon\}$.

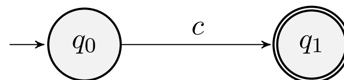

Figure 4.5: An NFA recognizing $\{c\}$.

for $A \cup B$? Well, a string is in $A \cup B$ if it is in either A or B, so this new NFA $N_{A\cup B}$ should accept a string x if and only if x is accepted by either N_A or N_B.

We construct such an NFA as follows. First, the alphabet of $N_{A\cup B}$ will simply be Σ. The states of $N_{A\cup B}$ will be all the states of both N_A and N_B: we will ensure these sets of sets are disjoint (renaming the states if necessary), and then take their union. Actually, $N_{A\cup B}$ will have one additional new start state q_0 (we will ensure this name is distinct from the state names in N_A and N_B by renaming those states if necessary).

The trick is this: the NFA $N_{A\cup B}$ will have an ϵ-transition from q_0 to the start state of N_A, and another ϵ-transition from q_0 to the start state of N_B. This will let the NFA choose which path it wants to take: a path that simulates N_A (and accepts the input if and only if $x \in A$), or a path that simulates N_B (and accepts the input if and only if $x \in B$). Since NFAs accept a string so long as *some* path reaches an accept state, this will mean $N_{A\cup B}$ will accept the strings that are *either* in A or in B. We will just need to set the accept states of $N_{A\cup B}$ to be the union of the accept states of N_A and N_B; the new state q_0 will not be an accept state (why not? Because otherwise it will always be the case that $N_{A\cup B}$ accepts ϵ, even if $\epsilon \notin A$ and $\epsilon \notin B$). All the transitions of $N_{A\cup B}$ will be just like the transitions of N_A and N_B, plus the two new ϵ-transitions.

This shows that if two languages A and B over an alphabet Σ have an NFA recognizing them, then so does the language $A \cup B$. What about the language AB? How might we construct an NFA for that?

Just as before, we will start with the NFAs N_A and N_B, and construct an NFA N_{AB} that has all the states of both (ensuring they have disjoint names). However, this time we connect the two NFAs differently. We would like the new NFA N_{AB} to accept a string x if and only if it has some decomposition into the concatenation of two strings, $x = yz$, such that y is accepted by N_A and z is accepted by N_B. To do so, the accepting paths of N_{AB} need to correspond to paths that reach an accept state of N_A, then switch to the start state of N_B, keep reading the input, and reach the accepting state of N_B at the end of the input string. How can we construct N_{AB} to force its accepting paths to do this?

The trick will be to set the start state of N_{AB} to the start state of N_A, and then to place ϵ-transitions from each the accept state of N_A to the start state of N_B. We will also set the accept states of N_{AB} to be the accept states of N_B, but *not* those of N_A.

To see why this works, suppose we have a string $x \in AB$. Then x can be written $x = yz$, with $y \in A$ and $z \in B$. Now, since N_A accepts y, there is some path in N_A from the start state of N_A to some accept state which reads exactly the input y; similarly, there is some path from the start state of N_B to some accept state of N_B which reads exactly the input z. Now, in N_{AB}, we can first follow the path in N_A that reads y and gets to an accept state of N_A; then, take the ϵ-transition to the start state of N_B; and finally, follow the path in N_B that reads z and reaches an accept state of N_B. Together, this forms a path in N_{AB} that reads x and reaches an accept state, meaning that

N_{AB} accepts the string $x \in AB$.

What about the other direction? We should verify that N_{AB} accepts *only* the strings in AB, not merely all of them. This is important, because to recognize AB, N_{AB} must accept *exactly* the strings in AB and no others. To do this, let x be an arbitrary string accepted by N_{AB}. This means that there is a path in N_{AB} that reaches an accept state and reads exactly x. Since the only accept states of N_{AB} are the accept states of N_B, and since we start at the start state of N_A, this accepting path for x must use an ϵ-transition to transition from the states of N_A to the states of N_B. Since we can only go from N_A to N_B (but not backwards), this transition point is unique: it happens exactly once in this accepting path. Let y be the string of all symbols of x read before this special ϵ-transition, and let z be the string of all symbols read afterwards. Then $x = yz$. It is also clear that there is a path in N_A that reads y and reaches an accept state of N_A (since reaching such a state is necessary to take the ϵ-transition), and similarly, there is a path in N_B that reads z and reaches an accept state of N_B. Hence N_A accepts y and N_B accepts z, so $y \in A$, $z \in B$, and thus $x = yz \in AB$. This means N_{AB} only accepts strings in AB, as desired.

Finally, we handle the star operation. Let A be a language, and let N_A be an NFA recognizing A. We want to construct an NFA N_{A^*} recognizing A^*. This construction is as follows: N_{A^*} will have all the states of N_A, plus an additional special start state q_0 (we will ensure this is different from the states of N_A by renaming those states if necessary). This special start state q_0 will have an ϵ-transition to the start state of N_A, and also, each accept state of N_A will have an ϵ-transition to q_0. Further, the new start state q_0 will be the only accept state of N_{A^*}; all the accept states of N_A will not be accept states in N_{A^*} (though they will each have ϵ-transitions to q_0).

Why does this work? We now show $L(N_{A^*}) = A^*$. First, let x be a string in A^*. Then either $x = \epsilon$, or else x can be written $x = y_1 y_2 \dots y_n$ for some $n \geq 1$ and some $y_1, y_2, \dots, y_n \in A$. If $x = \epsilon$, then N_{A^*} accepts x, because its start state is an accept state. Otherwise, we know that each y_i is accepted by N_A, and so there is some path from the start state of N_A to an accept state while reading y_i; now, in N_{A^*}, we can follow each of these paths in turn, preceeded by the ϵ-transition from q_0 to the start state of N_A and followed by taking the ϵ-transition back to q_0 once we complete each of the paths. This forms a path in N_{A^*} that starts and ends at the start state q_0 while reading $x = y_1 y_2 \dots y_n$; this is an accepting path for x, so N_{A^*} must accept each string $x \in A^*$.

What we showed so far is that $A^* \subseteq L(N_{A^*})$. For the other direction, suppose x is a string accepted by N_{A^*}. Then there is some accepting path in N_{A^*} that starts at q_0, ends at q_0 (the only accept state), and reads exactly x along the way. Now, split this path into sub-paths, with the split happening every time the path reenters the state q_0. If the path never leaves q_0, then $x = \epsilon$, which is always in A^* for each language A. Otherwise, these sub-paths read strings $y_1, y_2, \dots y_n$, with $n \geq 1$ and $y_1 y_2 \dots y_n = x$. Each y_i is read by some path in N_{A^*} of nonzero length that starts and ends at q_0, but otherwise does not visit q_0 except at the beginning and end. Since this path has nonzero length, it must leave q_0 at the beginning using the ϵ-transition to the start state of N_A; and since the path comes back to q_0 at the end, it must reach an accept state of N_A and take the ϵ-transition back to q_0. Removing the ϵ-transitions from the beginning and end leaves us with a path in N_A that reads exactly y_i, starts at the start state of N_A, and ends at an accept state of N_A. This means that N_A accepts the string y_i. We conclude that each y_i is in A, so $x = y_1 y_2 \dots y_n$ is in A^*, and hence $L(N_{A^*}) \subseteq A^*$, as desired.

We have now seen that the languages $\varnothing$, $\{\epsilon\}$, and $\{c\}$ for $c \in \Sigma$ all have NFAs recognizing them, and also that the concatenation, union, and star of languages recognized by NFAs can also be recognized by NFAs. Since regular expressions define languages by constructing them out of these basic operations, it follows that every language represented by a regular expression can be recognized by an NFA.

4.3 From NFAs to regular expressions

So far, we have seen how to convert between NFAs and DFAs (in both directions), and how to convert regular expressions into NFAs. The final step in showing they are all equivalent is to show how to convert NFAs into regular expressions.

We won't do this proof extremely formally, but we will go over the idea. The first thing we will do is generalize NFAs into *regular-expression* NFAs, which we call RNFAs. These will be similar to normal NFAs, except that their arcs will be labeled by *regular expressions* rather than by individual symbols. An example of an RNFA is given in Figure 4.6.

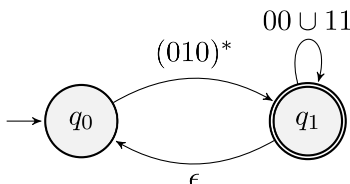

Figure 4.6: An example of an RNFA.

RNFAs are nondeterministic, just like NFAs; this means that there may be more than one viable path to follow for a given string. An RNFA is said to accept a string if there is any viable path that reads the string and reaches an accept state at the end.

How do we interpret arcs labeled by regular expressions? Well, such arcs can "match" to any sequence of symbols generated by that regular expression. For example, in the RNFA in Figure 4.6, the regular expression $(010)^*$ can match any of the following strings: ϵ, 010, 010010, 010010010, and so on. For instance, this RNFA accepts the string ϵ, because one possible path is to go from q_0 to q_1 without reading any symbols, and q_1 is an accept state. This RNFA also accepts the string 0101111, because we can go from q_0 to q_1 while reading 010, then loop on q_1 reading 11, and then loop again reading 11. On the other hand, this RNFA does not accept the string 0100. That's because there is no path that reads exactly 0100 and ends up at q_1. To see this, consider what an accepting path might look like for 0100. We could first read 010 and move from q_0 to q_1. But then we'll be at q_1 with only the string 0 left to read, and that has no matches except the ϵ-transition going back to q_0, from where we'll be forced to loop between q_0 and q_1 forever. There is no accepting path here, because an accepting path must read the *entire string*, and we will never be able to read the last 0 character.

Now that we have (informally) defined RNFAs, we will use them to turn regular NFAs into regular expressions by first viewing the NFA as an RNFA, and then repeatedly simplifying the RNFA by removing states.

Let N be an NFA. We will start by doing a bit of cleanup: first, we will add a new start state q_0, which we will ensure is different from the other states of N by renaming them if necessary. We will then add an ϵ-transition from q_0 to the former start state of N, so that the behavior of N doesn't change. This state q_0 will not be an accept state and will have no other transitions. (Why did we do this? Because it will be nice to assume that the start state cannot be returned to and is not an accept state, which we've now guaranteed.)

The second bit of cleanup is that we will add a new accept state p (again ensuring that it is different from all the other states by renaming them if necessary). This state p will be an accept state, and we will add an ϵ-transition from each former accept state of N to p. We will also make those former accept states no longer be accept states. This way, there will only be a single accepting state p, and once the NFA enters p is can no longer leave (except to drop into the abyss of always

rejecting). The state p will not have any loops or outgoing transitions on it. This means that every accepting path of our modified NFA N' will start at q_0 and end at p, and in the middle, it will never visit q_0 or p. Note that $L(N') = L(N)$; our modification so far did not affect which strings the NFA accepts.

We have now added two new states to the NFA. The next thing we will do is to view this NFA as an RNFA, and then remove all states from this RNFA one by one until only q_0 and p remain. At that point, our RNFA will have the form in Figure 4.7.

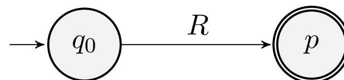

Figure 4.7: An RNFA with only the states q_0 and p remaining. Here R is some regular expression.

Note that if we manage to get an RNFA into this form which recognizes the same language as N, then it must be the case that $L(R) = L(N)$, because this RNFA accepts a string if and only if it is generated by R. Therefore, once we achieve our goal of removing all states from the RNFA N' except for q_0 and p, we will have converted our NFA N into a regular expression with the same language, just like we wanted.

To remove all these states from the RNFA, we just need to show how to remove a single state from an RNFA; we can then apply that procedure repeatedly. Moreover, we may assume that the state we wish to remove is not the start state and not an accept state.

How do we remove such a state q from an RNFA? Well, to maintain the behavior of the RNFA, we will need to make sure that all paths that pass through q get correctly rerouted elsewhere. To do so, we look at all pairs (q_1, q_2) of states that are not equal to q and such that there is an arc $q_1 \to q$ and another arc $q \to q_2$. For each such pair (q_1, q_2), we need to worry about how a path going from q_1 to q to q_2 will be implemented once we remove q. A general case of this situation is given in Figure 4.8.

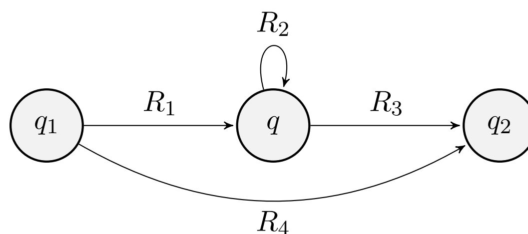

Figure 4.8: We wish to remove q while preserving possible paths from q_1 to q_2 passing through q. Here R_1, R_2, R_3, and R_4 are regular expressions.

To preserve all the possible paths passing from q_1 to q_2 through q, we will delete q (and the arcs labeled R_1, R_2, and R_3), delete the arc labeled R_4 (which may not even exist in the first place), and a new arc from q_1 to q_2 labeled $(R_1(R_2)^*R_3) \cup R_4$ (if R_4 doesn't exist, we can just label this $R_1(R_2)^*R_3$). It should be intuitive that this new arc allows the same strings to go from q_1 directly to q_2 as previously could go from q_1 to q_2 via either the direct arc or via q. Note that we do this for every pair of nodes (q_1, q_2) that are not equal to q and are connected to q in this way; this means we will be adding or modifying a large number of arcs to delete the single node q.

This same construction still works when q_1 is the start state q_0 or when q_2 is the accept state p. It even works when $q_1 = q_2$; in that case, the arc from q_1 to q_2 will be a loop, but everything else will work the same.

By repeatedly deleting the intermediate nodes (those note equal to q_0 or p) in this way, we can reduce the number of states down to only two: q_0 and p. Those states don't have self-loops, so we can therefore turn any NFA N into an RNFA of the form in Figure 4.7 for some regular expression R generating the language $L(N)$, as desired.

4.4 Next steps

We have now seen how to convert between DFAs, NFAs, and regular expressions. We've shown that all three can recognize the same class of languages, call the regular languages. Next lecture, we will show a few more properties of regular languages, and then start talking about which types of languages are *not* regular, and how to prove it.