CS 373: Theory of Computation
Sariel Har-Peled and Madhusudan Parthasarathy

Lecture 8: From DFAs/NFAs to Regular Expressions

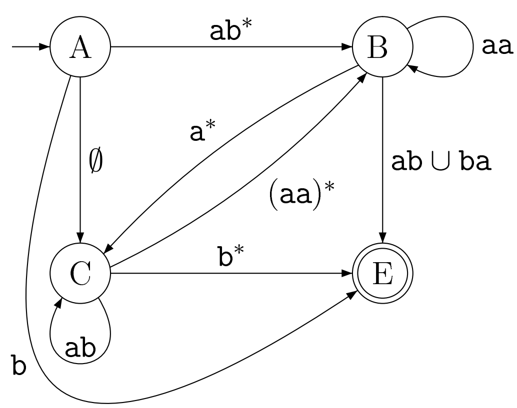
12 February 2009

In this lecture, we will show that any DFA can be converted into a regular expression. Our construction would work by allowing regular expressions to be written on the edges of the DFA, and then showing how one can remove states from this generalized automata (getting a new equivalent automata with the fewer states). In the end of this state removal process, we will remain with a generalized automata with a single initial state and a single accepting state, and it would be then easy to convert it into a single regular expression.

1 From NFA to regular expression

1.1 GNFA— A Generalized NFA

Consider an NFA N where we allowed to write any regular expression on the edges, and not only just symbols. The automata is allowed to travel on an edge, if it can matches a prefix of the unread input, to the regular expression written on the edge. We will refer to such an automata as a ***GNFA*** (***generalized non-deterministic finite automata*** [Don't you just love all these shortcuts?]).



Thus, the GNFA on the right, accepts the string abbbbaaba, since

$$A \xrightarrow{\texttt{abbbb}} B \xrightarrow{\texttt{aa}} B \xrightarrow{\texttt{ba}} E.$$

To simplify the discussion, we would enforce the following conditions:

(C1) There are transitions going from the initial state to all other states, and there are no transitions into the initial state.

(C2) There is a single accept state that has only transitions coming into it (and no outgoing transitions).

(C3) The accept state is distinct from the initial state.

(C4) Except for the initial and accepting states, all other states are connected to all other states via a transition. In particular, each state has a transition to itself.

When you can not actually go between two states, a GNFA has a transitions labelled with $\emptyset$, which will not match any string of input characters. We do not have to draw these transitions explicitly in the state diagrams.

1.2 Top-level outline of conversion

We will convert a DFA to a regular expression as follows:

(A) Convert DFA to a GNFA, adding new initial and final states.

(B) Remove all states one-by-one, until we have only the initial and final states.

(C) Output regex is the label on the (single) transition left in the GNFA. (The word ***regex*** is just a shortcut for regular expression.)

Lemma 1.1 *A DFA M can be converted into an equivalent GNFA G.*

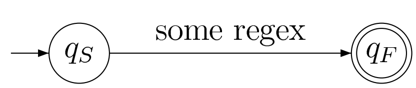
Proof: We can consider M to be an NFA. Next, we add a special initial state q_{init} that is connected to the old initial state via ϵ-transition. Next, we add a special final state q_{final}, such that all the final states of M are connected to q_{final} via an ϵ-transition. The modified NFA M' has an initial state and a single final state, such that no transition enters the initial state, and no transition leaves the final state, thus M' comply with conditions (C1–C3) above. Next, we consider all pair of states $x, y \in Q(M')$, and if there is no transition between them, we introduce the transition $x \xrightarrow{\emptyset} y$. The resulting GNFA G from M' is now compliant also with condition (C4).

It is easy now to verify that G is equivalent to the original DFA M. ∎

We will remove all the intermediate states from the GNFA, leaving a GNFA with only initial and final states, connected by one transition with a (typically complex) label on it. The equivalent regular expression is obvious: the label on the transition.

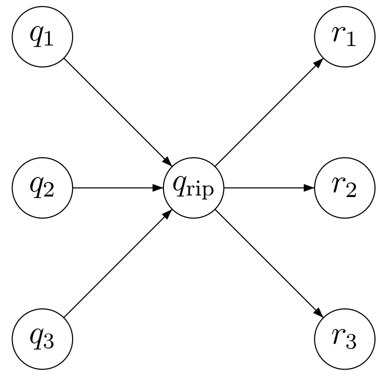
Lemma 1.2 *Given a GNFA N with $k = 2$ states, one can generate an equivalent regular expression.*

Proof: A GNFA with only two states (that comply with conditions (C1)-(C4)) have the following form.



The GNFA has a single transition from the initial state to the accepting state, and this transition has the regular expression R associated with it. Since the initial state and the accepting state do not have self loops, we conclude that N accepts all words that matches the regular expression R. Namely, $L(N) = L(R)$. ∎

1.3 Details of ripping out a state



We first describe the construction. Since $k > 2$, there is at least one state in N which is not initial or accepting, and let q_{rip} denote this state. We will "rip" this state out of N and fix the GNFA, so that we get a GNFA with one less state.

Transition paths going through q_{rip} might come from any of a variety of states q_1, q_2, etc. They might go from q_{rip} to any of another set of states r_1, r_2, etc.

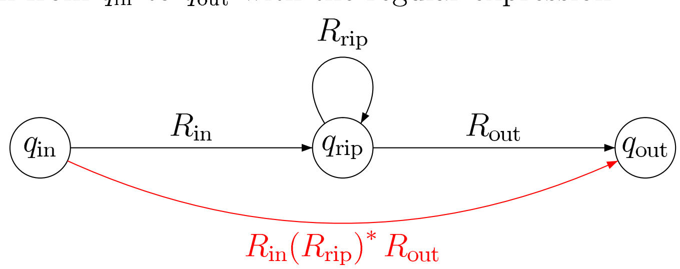
For each pair of states q_i and r_i, we need to convert the transition through q_{rip} into a direct transition from q_i to r_i.

1.3.1 Reworking connections for specific triple of states

To understand how this works, let us focus on the connections between q_{rip} and two other specific states q_{in} and q_{out}. Notice that q_{in} and q_{out} might be the same state, but they both have to be different from q_{rip}.

The state q_{rip} has a self loop with regular expression R_{rip} associated with it. So, consider a fragment of an accepting trace that goes through q_{rip}. It transition into q_{rip} from a state q_{in} with a regular expression R_{in} and travels out of q_{rip} into state q_{out} on an edge with the associated regular expression being R_{out}. This trace, corresponds to the regular expression R_{in} followed by 0 or more times of traveling on the self loop (R_{rip} is used each time we traverse the loop), and then a transition out to q_{out} using the regular expression R_{out}. As such, we can introduce a direct transition from q_{in} to q_{out} with the regular expression

$$R = R_{\text{in}}(R_{\text{rip}})^*\, R_{\text{out}}.$$



Clearly, any fragment of a trace traveling $q_{\text{in}} \to q_{\text{rip}} \to q_{\text{out}}$ can be replaced by the direct transition $q_{\text{in}} \xrightarrow{\texttt{R}} q_{\text{out}}$. So, let us do this replacement for any two such stages, we connect them directly via a new transition, so that they no longer need to travel through q_{rip}.

Clearly, if we do that for all such pairs, the new automata accepts the same language, but no longer need to use q_{rip}. As such, we can just remove q_{rip} from the resulting automata. And let M' denote the resulting automata.

The automata M' is not quite legal, yet. Indeed, we will have now parallel transitions because of the above process (we might even have parallel self loops). But this is easy to fix: We replace two such parallel transitions $q_i \xrightarrow{\texttt{R}_1} q_j$ and $q_i \xrightarrow{\texttt{R}_2} q_j$, by a single transition

$$q_i \xrightarrow{\texttt{R}_1+\texttt{R}_2} q_j.$$

As such, for the triple $q_{\text{in}}, q_{\text{rip}}, q_{\text{out}}$, if the original label on the direct transition from q_{in} to q_{out} was originally R_{dir}, then the output label for the new transition (that skips q_{rip}) will be

$$R_{dir} \ + \ R_{\text{in}}(R_{\text{rip}})^*\, R_{\text{out}}. \tag{1}$$

Clearly the new transition, is equivalent to the two transitions it replaces. If we repeat this process for all the parallel transitions, we get a new GNFA M which has $k-1$ states, and furthermore it accepts exactly the same language as N.

1.4 Proof of correctness of the ripping process

Lemma 1.3 *Given a GNFA N with $k > 2$ states, one can generate an equivalent GNFA M with $k-1$ states.*

Proof: Since $k > 2$, N contains least one state in N which is not accepting, and let q_{rip} denote this state. We will "rip" this state out of N and fix the GNFA, so that we get a GNFA with one less state.

For every pair of states q_{in} and q_{out}, both distinct from q_{rip}, we replace the transitions that go through q_{rip} with direct transitions from q_{in} to q_{out}, as described in the previous section.

Correctness. Consider an accepting trace T for N for a word w. If T does not use the state q_{rip} than the same trace exactly is an accepting trace for M. So, assume that it uses q_{rip}, in particular, the trace looks like

$$T = \ldots q_i \xrightarrow{\mathtt{S_i}} q_{\mathrm{rip}} \overbrace{\xrightarrow{\mathtt{S_{i+1}}} q_{\mathrm{rip}} \ldots \xrightarrow{\mathtt{S_{j-1}}} q_{\mathrm{rip}}}^{\text{0 or more times}} \xrightarrow{\mathtt{S_{j-1}}} q_j \ldots.$$

Where $S_i S_{i+1} \ldots, S_j$ is a substring of w. Clearly, $S_i \in R_{\mathrm{in}}$, where R_{in} is the regular expression associated with the transition $q_i \to q_{\mathrm{rip}}$. Similarly, $S_{j-1} \in R_{\mathrm{out}}$, where R_{out} is the regular expression associated with the transition $q_{\mathrm{rip}} \to q_j$. Finally, $S_{i+1} S_{i+2} \cdots S_{j-1} \in (R_{\mathrm{rip}})^*$, where R_{rip} is the regular expression associated with the self loop of q_{rip}.

Now, clearly, the string $S_i S_{i+1} \ldots S_j$ matches the regular expression $R_{\mathrm{in}}(R_{\mathrm{out}})^* R_{\mathrm{out}}$. in particular, we can replace this portion of the trace of T by

$$T = \ldots q_i \xrightarrow{\mathtt{S_i S_{i+1} \ldots S_{j-1} S_j}} q_j \ldots.$$

This transition is using the new transition between q_i and q_j introduced by our construction. Repeating this replacement process in T till all the appearances of q_{rip} are removed, results in an accepting trace $\widehat{T}$ of M. Namely, we proved that any string accepted by N is also accepted by M.

We need also to prove the other direction. Namely, given an accepting trace for M, we can rewrite it into an equivalent trace of N which is accepting. This is easy, and done in a similar way to what we did above. Indeed, if a portion of the trace uses a new transition of M (that does not appear in N), we can place it by a fragment of transitions going through q_{rip}. In light of the above proof, it is easy and we omit the straightforward but tedious details. ■

Theorem 1.4 *Any DFA can be translated into an equivalent regular expression.*

Proof: Indeed, convert the DFA into a GNFA N. As long as N has more than two states, reduce its number of states by removing one of its states using Lemma **??**. Repeat this process till N has only two states. Now, we convert this GNFA into an equivalent regular expression using Lemma **??**. ■

1.5 Running time

This is a relatively inefficient algorithm. Nevertheless, it establishes the equivalence between the automata and regular expressions. Fortunately, it is a conversion that you rarely need to do in practical applications. Usually, the input would be the regex and the application would convert it *into* an NFA or DFA. Converting in that direction is more efficient.

To realize the problem, note that the algorithm for ripping a single state has three nested loops in it.

For every state q_{rip} **do**

For every incoming state q_{in} **do**

For every outgoing state q_{out} **do**

Remove all transition paths from q_{in} to q_{out} via q_{rip} by creating a direct transition between q_{in} and q_{out}.

So, if the original DFA has n states, then the algorithm will do the inner step $O(n^3)$ times (which is not too bad). Worse, each time we remove a state, we replace the regex on each remaining transition with a regex that is potentially four times as large. (That is, we replace the regular expression R_{dir} associated with a transition, by a regular expression $R_{dir} + R_{\text{in}}(R_{\text{rip}})^* R_{\text{out}}$, see Eq. (**??**)$_{\text{p??}}$.)

So, every time we rip a state in the GNFA, the length of the regular expressions associated with the edges of th GNFA get longer by a factor of four (at most). So, we repeat this n times, so the length of the final output regex is $O(4^n)$. And the actual running time of the algorithm is $O(n^3 4^n)$.

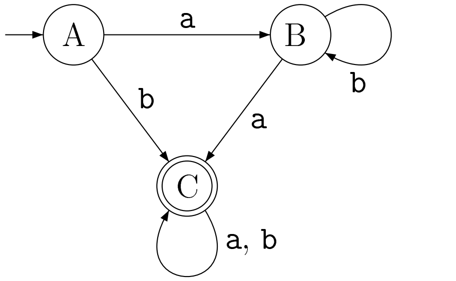
Typically output sizes and running times are not quite that bad. We really only need to consider triples of states that are connected by arcs with labels other than $\emptyset$. Many transitions are labelled with ϵ or $\emptyset$, so regular expression size often increases by less than a factor of 4. However, actual times are still unpleasant for anything but very small examples.

Interestingly, while this algorithm is not very efficient, it is not the algorithm "fault". Indeed, it is known that regular expressions for automata can be exponentially large. There is a lower bound of 2^n for regular expressions describing an automata of size n, see [**?**] for details.

2 Examples

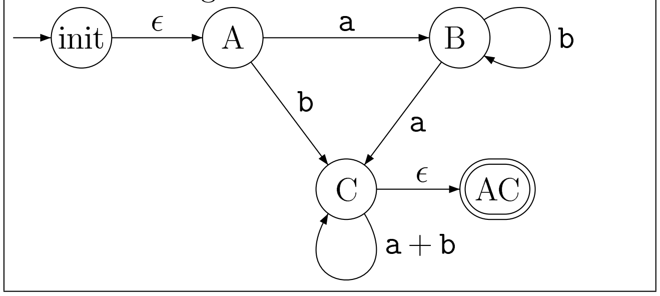
2.1 Example: From GNFA to regex in 8 easy figures

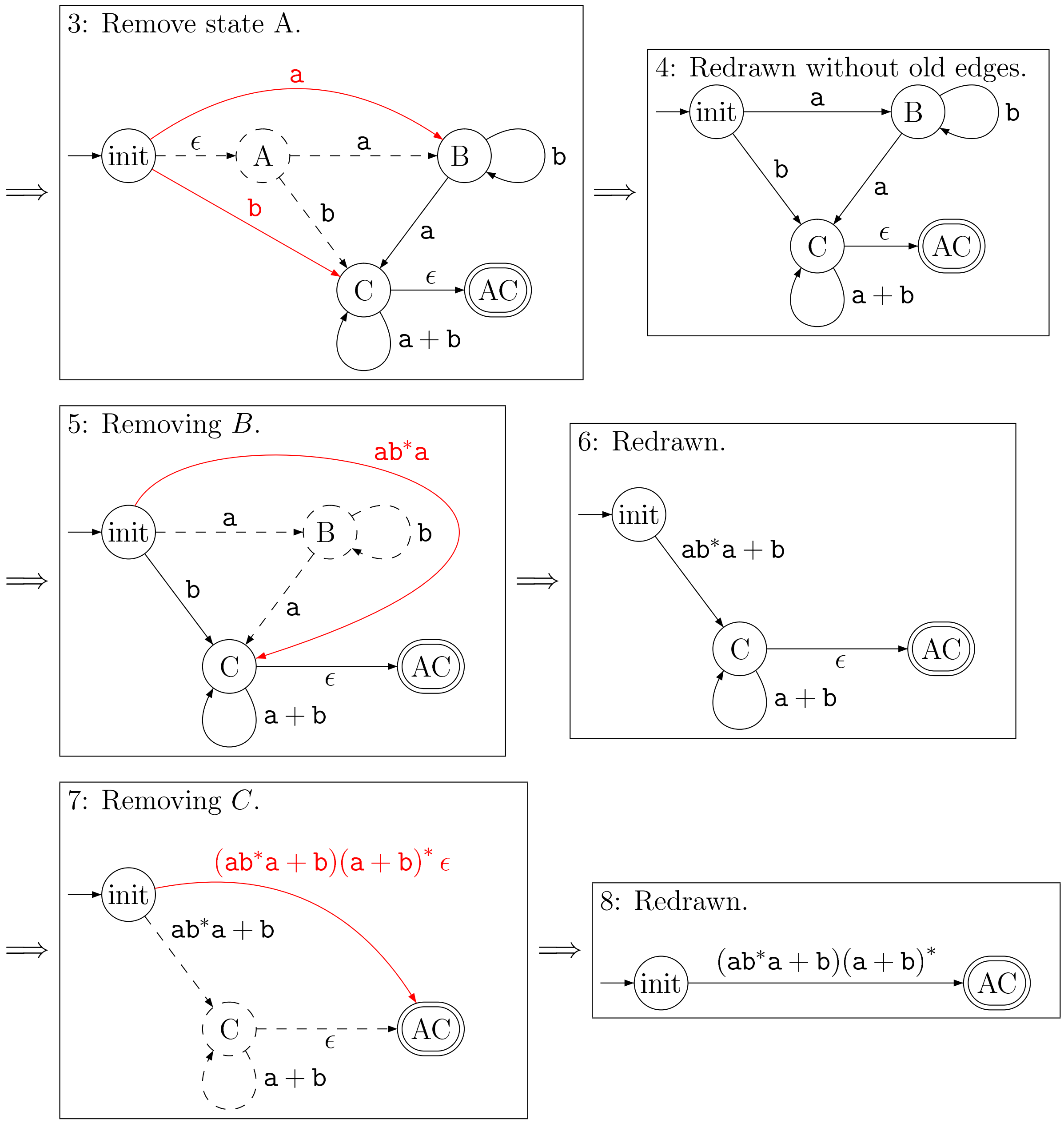
1: The original NFA.



$\Longrightarrow$

2: Normalizing it.





Thus, this automata is equivalent to the regular expression $(\mathtt{ab}^*\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b})^*$.

3 Closure under homomorphism

Suppose that Σ and Γ are two alphabets (possibly the same, but maybe different). A ***homomorphism*** h is a function from Σ^* to Γ^* such that $h(xy) = h(x)h(y)$ for any strings x and y. Equivalently, if we divide w into a sequence of individual characters $w = c_1c_2 \ldots c_k$, then $h(w) = h(c_1)h(c_2) \ldots h(c_k)$. (It's a nice exercise to prove that the two definitions are equivalent.)

Example 3.1 Let $\Sigma = \{a, b, c\}$ and $\Gamma = \{0, 1\}$, and let h be the mapping $h : \Sigma \to \Gamma$, such that $h(a) = 01$, $h(b) = 00$, $h(c) = \epsilon$. Clearly, h is a homomorphism.

So, suppose that we have a regular language L. If L is represented by a regular expression R, then it is easy to build a regular expression for $h(L)$. Just replace every character c in R by its image $h(c)$.

Example 3.2 The regular expression $R = (ac + b)^*$ over Σ becomes $h(R) = (01 + 00)^*$.

Lemma 3.3 *Let L Be a regular language over Σ, and let $h : \Sigma \to \Gamma$ be a homomorphism, then the language $h(L)$ is regular.*

Proof: (Informal.) Let R Be a regular expression for R. Replace any character $c \in \Sigma$ appearing in R by the string $h(c)$. Clearly, the resulting regular expression R' recognizes all the words in $h(L)$. ■

Proof:(More formal.) Let $\mathtt{M}$ be a NFA for L with a single accept state q_{final} and an initial state q_{init}, so that the only transitions from q_{init} is ϵ-transition out of it, and the is no outgoing transitions from q_{final} and only ϵ-transitions into it.

Now, replace every transition $q \xrightarrow{\mathtt{c}} q'$ in $\mathtt{M}$ by the transition $q \xrightarrow{\mathtt{h(c)}} q'$. Clearly, the resulting automata is a GNFA $\mathtt{D}$ that accepts the language $h(L)$. We showed in the previous lecture, that a GNFA can be converted into an equivalent regular expression R, such that $L(\mathtt{D}) = h(R)$. As such, we have that $h(L) = L(\mathtt{D}) = h(R)$. Namely, $h(L)$ is a regular language, as claimed. ■

Note, that in the above proof, instead of creating a GNFA, we can also create a NFA, by introducing temporary states. Thus, if we have the transition $q \xrightarrow{\mathtt{c}} q'$ in $\mathtt{M}$, and $h(c) = w_1 w_2 \ldots w_k$, then we will introduce new temporary states $s_1, \ldots s_{k-1}$, and replace the transition $q \xrightarrow{\mathtt{c}} q'$ by the transitions

$$q \xrightarrow{\mathtt{w_1}} s_1,\ s_1 \xrightarrow{\mathtt{w_2}} s_2,\ \ldots\ s_{k-2} \xrightarrow{\mathtt{w_{k-1}}} s_{k-1},\ s_{k-1} \xrightarrow{\mathtt{w_k}} q'.$$

Thus, we replace the transition $q \xrightarrow{\mathtt{c}} q'$ by a path between q and q' that accepts only the string $h(c)$. It is now pretty easy to argue that the language of the resulting NFA $\mathtt{D}$ is $h(L)$.

Note that when you have several equivalent representations, do your proofs in the one that makes the proof easiest. So we did set complement using DFAs, concatenation using NFAs, and homomorphism using regular expressions. Now we just have to finish the remaining bits of the proof that the three representations are equivalent.

An interesting point is that if a language L is note regular then $h(L)$ might be regular or not.

Example 3.4 Consider the language $L = \left\{ \mathtt{a}^n \mathtt{b}^n \,\middle|\, n \geq 0 \right\}$. The language L is not regular. Now, consider the homomorphism $h(\mathtt{a}) = \mathtt{a}$ and $h(\mathtt{b}) = \mathtt{a}$. Clearly, $h(L) = \left\{ \mathtt{a}^n \mathtt{a}^n = \mathtt{a}^{2n} \,\middle|\, n \geq 0 \right\}$, which is definitely regular. However, the identify homomorphism $I(\mathtt{a}) = \mathtt{a}$ and $I(\mathtt{b}) = \mathtt{b}$ maps L to itself $I(L) = L$, and as such $I(L)$ is not regular.

Intuitively, homomorphism can not make a language to be "harder" than it is (if it is regular, then it remains regular under homomorphism). However, if it is not regular, it might remain not regular.

References

[EZ74] A. Ehrenfeucht and P. Zeiger. Complexity measures for regular expressions. In *Proc. 6th Annu. ACM Sympos. Theory Comput.*, pages 75–79, 1974. `http://portal.acm.org/citation.cfm?id=803886`.