ECS 120 Lesson 5 – Nondeterministic Finite State Machines

Oliver Kreylos

Monday, April 9th, 2001

Before we prove the closure of regular languages under concatenation and Kleene Star, we introduce another type of finite automaton that will make the proofs a lot easier.

1 Nondeterministic Finite State Machines

Until now, the transition function δ for an automaton $M = (Q, \Sigma, \delta, q_0, F)$ was restricted by requiring that for each state $q \in Q$ and every character $a \in \Sigma$, there was exactly one other state $q' \in Q$ such that $\delta(q, a) = q'$. This made δ a total function. We will henceforth call machines of this restricted type *deterministic finite automata (DFA)*. A new type of machine, called *nondeterministic finite automaton (NFA)*, is created by dropping the restriction on δ. There are three differences between the transition function of an NFA and that of a DFA, see Figure 1:

- There can be states with more than one arrow leaving for the same input symbol (see state q_0 and symbol `1` in Fig. 1).
- There can be states with no arrows leaving for an input symbol (see state q_2 and symbol `0` in Fig. 1).
- There can be arrows labelled with the special symbol ϵ (see state q_1 in Fig. 1).

How does the computation of an NFA differ from that of a DFA? To recall, a DFA computes by starting in its start state, and following a chain of

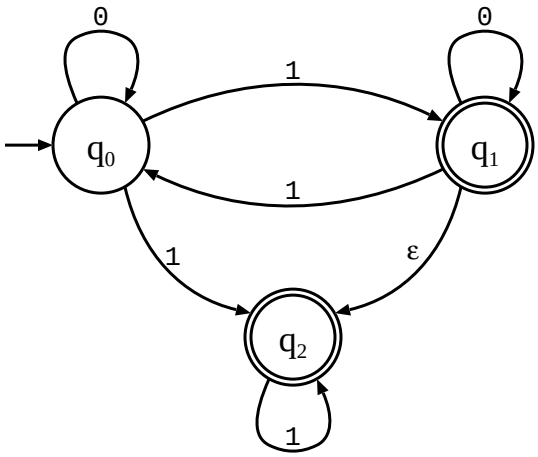

Figure 1: An NFA M over the alphabet $\Sigma = \{0, 1\}$.

states labeled by the sequence of characters read, see Figure 2. An NFA, on the other hand, computes by following a *tree* of state chains starting at its start state, see Figure 3. Here are the three ways in which NFA computation differs from DFA computation:

- If the machine reads a character with multiple arrows leaving from the current state, it will branch the chain of states and follow all those arrows simultaneously.
- If the machine reads a character with no arrows leaving from the current state, the chain of states will "die," i. e., it will terminate prematurely.
- If the machine is in a state with one or more leaving ϵ-arrows, it will branch the chain of states by following all those arrows simultaneously before reading any characters.

For a DFA, which only had a single chain of states, we defined that a word w is accepted iff the chain of states upon reading w ends in a final state. For an NFA, which has multiple chains, we acceptance of a word w is defined by requiring that at least one chain of states that is still "alive" ends in a final state.

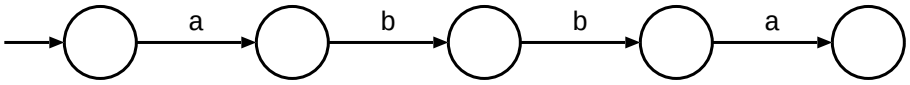

Figure 2: The computation of a DFA always forms a single chain of states, where the labels on the arrows spell out the input word.

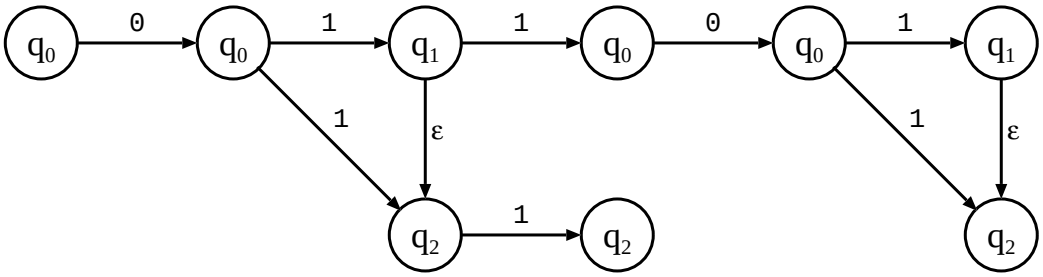

Figure 3: The computation tree of the automaton M from Fig. 1 upon reading the input word $w = \texttt{01101}$.

2 Formal Definition of NFAs

An NFA M is a five-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states.
- Σ is the input alphabet.
- The transition function $\delta\colon Q \times \big(\Sigma \cup \{\epsilon\}\big) \to \mathcal{P}(Q)$ is a function taking a state and either a character from Σ or the special symbol ϵ (which must not be an element of Σ) as input, and returns a set of states.
- $q_0 \in Q$ is the start state.
- $F \subset Q$ is the set of final states.

As mentioned earlier, the only difference from the formal definition of a DFA is the different transition function δ.

3 Formal Definition of Computation of NFAs

We will formally define the computation of an NFA by building its extended transition function δ^*, as we did for DFAs. But before we do that, we in-

troduce a helper function to deal with ϵ-transitions. If $q \in Q$ is a state, we define the *ϵ-closure of q*, the set of all states that can be reached from q by only performing ϵ-transitions, recursively as follows:

1. $q \in \text{ECLOSE}(q)$, i. e., any state can be reached from itself by only following ϵ-transitions – even if there are none.

2. Let $p \in \text{ECLOSE}(q)$ be some state in the ϵ-closure of q. If there exists an ϵ-transition from p, i. e., there exists a state $r \in Q$ such that $r \in \delta(p, \epsilon)$, we include the state r into the ϵ-closure of q: $r \in \text{ECLOSE}(q)$.

Furthermore, if $f\colon X \to \mathcal{P}(Y)$ is some function from a set X to a power set $\mathcal{P}(Y)$, and $A \subset X$ is a subset of X, we define the shorthand notation $f(A) := \bigcup_{x \in A} f(x)$. That is, we apply f to all elements $x \in A$, and unite all the resulting sets $f(x)$. Now we can define $\delta^*(q, w)$ recursively:

1. If $w = \epsilon$, i. e., w is the empty word, then $\delta^*(q, w) = \delta^*(q, \epsilon) := \text{ECLOSE}(q)$.

2. If $w = ax$ for some $a \in \Sigma$ and some $w \in \Sigma^*$, then $\delta^*(q, w) = \delta^*(q, ax) := \delta^*\big(\delta(\text{ECLOSE}(q), a), x\big)$. In other words, we first perform all possible ϵ-transitions from q by calculating the ϵ-closure of q, then apply the transition function with the character a to all the states in the ϵ-closure, and finally apply the extended transition function δ^* with the shortened word x to all the resulting states.

Using the extended transition function δ^*, we can define that an NFA $M = (Q, \Sigma, \delta, q_0, F)$ accepts a word $w \in \Sigma^*$, iff $\delta^*(q_0, w) \cap F \neq \emptyset$. This is equivalent to the informal definition that at least on chain of states in M's computation ends in a final state. Similarly, we define the language of NFA M as $L(M) := \big\{\, w \in \Sigma^* \;\big|\; \delta^*(q_0, w) \cap F \neq \emptyset \,\big\}$.

4 Equivalence of NFAs and DFAs

We introduced NFAs as a more powerful way of expressing finite state machines: By definition, every DFA is also an NFA; thus, the class of DFAs is a subset of the class of NFAs. For the same reason, the class of languages accepted by DFAs is a subset of the class of languages accepted by NFAs, or shorter, $L(\text{DFA}) \subset L(\text{NFA})$. The question now is: Are nondeterministic

automata really more powerful than their deterministic counterparts, i. e., are there languages that can not be decided by DFAs, but can be decided by NFAs? The answer to this question is, suprisingly, no. We will prove later that NFAs can be simulated by DFAs, thus, every language recognized by an NFA can also be recognized by a DFA. This situation can be compared to programming languages: Higher-level languages, e. g., C, C++ or Java, are more comfortable to use and more powerful in their expression than assembly language; however, any higher-level language program must be executed by a computer in the end and can therefore be expressed as an assembly program as well. The process to convert a higher-level language program to assembly is called *compilation*; here, we will use a comparable procedure to "compile" an NFA into an equivalent DFA.

5 The Subset Construction

The idea behind constructing a DFA equivalent to a given NFA is that, as opposed to a DFA, an NFA can be in several states simultaneously. The computational configuration of a DFA is defined uniquely by the one state it is currently in, whereas the configuration of an NFA is defined by the set of states it is in. Therefore, our idea is to construct a DFA that has all possible subsets of states of the given NFA as states. We will construct the new machine's transition function in such a way, that whenever the NFA is in a set of states $P \subset Q$, the DFA will be in the single state $P \in \mathcal{P}(Q)$.

Let A be a language accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$. Then we define a deterministic automaton $D = (Q_D, \Sigma, \delta_D, q_{D0}, F_D)$, where:

- The set of states $Q_D = \mathcal{P}(Q)$ is the power set – the set of all subsets – of the NFA's state set Q.
- $\delta_D\colon \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$ is defined for all states $P \in \mathcal{P}(Q)$ and all input characters $a \in \Sigma$ by $\delta_D(P, a) := \text{ECLOSE}\big(\delta(P, a)\big)$. Note that we are applying the functions ECLOSE and δ to sets of arguments, as defined above. Informally, the set $P' \subset Q$ of states, which is a single state $P' \in \mathcal{P}(Q)$ in the DFA D, returned by $\delta_D(P, a)$ is the set of all states that can be reached from any state in P by first applying a single transition on the input character a and then any number of ϵ-transitions. Figure 4 shows an illustration of how δ_D works.

- $q_{D0} \in \mathcal{P}(Q)$ is the set of all states that M can be in before reading any characters. If M had no ϵ-transitions from its start state q_0, q_{D0} would be the set consisting only of the start state, $q_{D0} = \{q_0\}$. To take possible ϵ-transitions into account, we define $q_{D0} := \text{ECLOSE}(q_0)$ instead.

- $F_D \subset \mathcal{P}(Q)$ is the set of all subsets of Q that contain at least one final state: $F_D := \left\{ P \subset Q \;\middle|\; P \cap F \neq \emptyset \right\}$. This way, whenever machine M ends in at least one final state in any of its branches of computation, D will end in a final state as well.

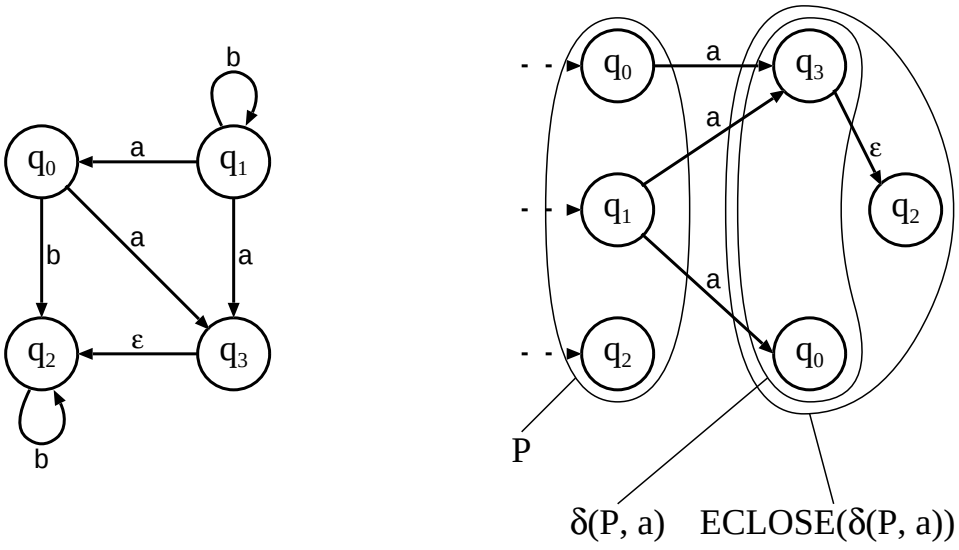

Figure 4: Definition of δ_D in the subset construction. Left: A fragment of an NFA; right: A fragment of a possible tree of state chains of M. At the beginning of the fragment, M is in states q_0, q_1 and q_2; thus, D is in state $P = \{q_0, q_1, q_2\}$. To calculate $\delta_D(P, a)$, we first follow all arrows labeled with the input symbol `a` leaving from any state $q \in P$ to find $\delta(P, a)$, and then calculate the ϵ-closure $\text{ECLOSE}\big(\delta(P, a)\big)$ of $\delta(P, a)$.