COMS W3261
Computer Science Theory
Chapter 2 Notes

Alexander Roth

2014-09-06

Definitions

Deterministic The automaton cannot be in more than one state at any time.

Nondeterministic The automaton may be in several states at once.

An Informal Picture of Finite Automata

We will be using the example of a real-world problem whose solution uses finite automata in an important role. We investigate protocols that support "electronic money" – files that a customer can use to pay for goods on the internet, and that the seller can receive insurance that the "money" is real.

The Ground Rules

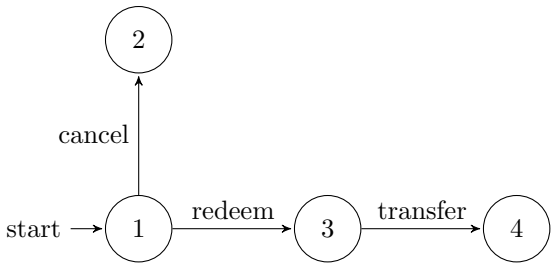
There are three participants: the customer, the store, and the bank. We assume for simplicity that there is only one "money" file in existence. Interactions among the three participants is thus limited to five elements:

1. The customer may decide to *pay*. That is, the customer sends the money to the store.
2. The customer may decide to *cancel*. The money is sent to the bank with a message that the value of the money is to be added to the customer's bank account.
3. The store may *ship* goods to the customer.
4. The store may *redeem* the money. That is, the money is sent to the bank with a request that its value be given to the store.
5. The bank may *transfer* the money by creating a new, suitably encrypted money file and sending it to the store.

The Protocol

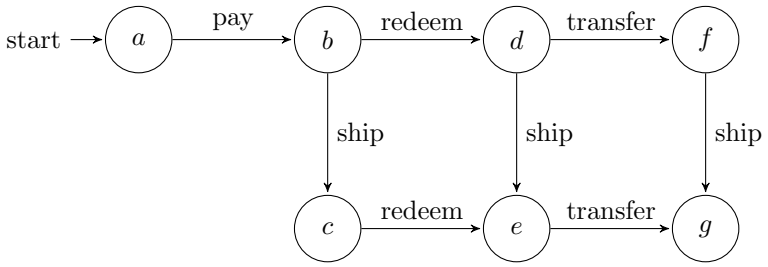
The customer cannot be relied upon to act responsibly. In particular, the customer may try to copy the money file, use it to pay several times, or both pay and cancel the money, thus getting the goods "for free". The bank must behave responsibly, or it cannot be a bank. The store should be careful as well. In particular, it should not ship goods until it is sure it has been given valid money for the goods.

These types of protocols can be represented by finite automata. Each state represents a situation that one of the participants could be in. Transitions between states occur when one of the five events described above occur.



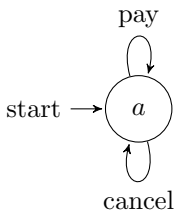
This is the automaton for the bank. The start state is state 1; it represents the situation where the bank has issued the money file in question but has not been requested either to redeem or cancel it. If a *cancel* request is sent to the bank by the customer, then the bank restores the money to the customer's account and enters state 2. The bank, being responsible, will not leave state 2 once it is entered, since the bank must not allow the same money to be cancelled again or spent by the customer. Similar concepts should be applied to the other states.

Now, let us consider the automaton of the actions for the store.



There are some defects in the store's design. Imagine that the shipping and financial operations are done by separate processes, so there is the opportunity for the *ship* action to be done either before, after, or during the redemption of the electronic money. Thus, the store can get into situations where it has already shipped the goods only to find out that the money was bogus.

Finally, there is the automaton for the customer.



This automaton only has one state, reflecting the fact that the customer "can do anything." The customer can perform the *pay* and *cancel* actions any number of times, in any order, and stays in the lone state after each action.

Enabling the Automata to Ignore Actions

While the three automata reflect the behaviors of the three participants independently, there are certain transitions that are missing. For example, the store is not affected by a *cancel* message. According to the formal definition of a finite automaton, whenever an input X is received by an automaton, the automaton must follow an arc labeled X from the state it is in to some new state.

Thus, in the current scenario, if the store were to receive a *cancel* action from the customer, the store automaton would "die"; that is, the automaton would be in no state at all, and further actions by that automaton would be impossible. This gives rise to the actions that must be ignored by an automaton:

1. *Actions that are irreleveant to the participant involved.*
2. *Actions that must not be allowed to kill the automaton.*

Deterministic Finite Automata

The term "deterministic" refers to the fact that on each input there is one and only one state to which the automaton can transition from its current state. In contrast, "nondeterministic" can be in several states at once.

Definition of a Deterministic Finite Automaton

A *deterministic finite automaton* consists of:

1. A finite set of *states*, often denoted by Q.
2. A finite set of *input symbols*, often denoted by Σ.
3. A *transition function* that takes as arguments a state and an input symbol and returns a state. The transition function will be commonly noted as δ. If q is a state, and a is an input symbol, then $\delta(q, a)$ is the state p such that there is an arc labeled a from q to p[2].
4. A *start state*, one of the states in Q.

5. A set of *final* or *accepting* states F. The set F is a subset of Q.

In proofs, we often talk about a DFA in "five-tuple" notation:

$$A = (Q, \Sigma, \delta, q_0, F)$$

where A is the name of the DFA, Q is its set of states, Σ its input symbols, δ its transition function, q_0 its start state, and F its set of accepting states.

How a DFA Processes Strings

The "language" of a DFA is the set of all strings that the DFA accepts. Suppose $a_1 a_2 \cdots a_n$ is a sequence of input symbols. We start our DFA in its start state, q_0. We consult the transition function δ, say $\delta(q_0, a_1) = q_1$ to find the state that the DFA A enters after processing the first input symbol a_1. We continue in this manner, finding states $q_2, q_3, \cdots, q_n$ such that $\delta(q_{i-1}, a_i) = q_i$ for each i. If q_n is a member of F, then the input $a_1 a_2 \cdots a_n$ is accepted, and if not then it is "rejected".

Simpler Notations for DFA's

There are two preferred notations for describing automata:

1. A *transition diagram*, which is a graph such as the ones construct earlier.
2. A *transition table*, which is a tabular listing of δ function, which by implication tells us the set of states and the input alphabet.

Transition Diagrams

A *transition diagram* for a DFA $A = (Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows:

a) For each state in Q there is a node.

b) For each state q in Q and each input symbol a in Σ, let $\delta(q, a) = p$. Then the transition diagram has an arc from node q to node p, labeled a. If there are several input symbols that cause transitions from q to p, then the transition diagram can have one arc, labeled by the list of these symbols.

c) There is an arrow into the start state q_0, labeled *Start*. This arrow does not originate at any node.

d) Nodes corresponding to accepting states (those in F) are marked by a double circle. States not in F have a single circle.

Transition Tables

A *transition table* is a conventional, tabular representation of a function like δ that takes two arguments and returns a value. The rows of the table correspond to the states, and the columns correspond to the input. The entry for the row corresponding to state q and the column corresponding to input a is the state $\delta(q, a)$. The start state is marked with an arrow, and the accepting state is marked with a star.

Extending the Transition Function to Strings

We need to make the notion of the language of a DFA precise. Thus, we define an *extended transition function* that describes what happens when we start in any state and follow any sequence of inputs. If δ is our transition function, then the extended transition function constructed from δ will be called $\hat{\delta}$. The extended transition function is a function that takes a state q and a string w and returns a state p – the state that the automaton reaches when starting in state q and processing the sequence of inputs w.

The Language of a DFA

Now, we can define the *language* of a DFA $A = (Q, \Sigma, \delta, q_0, F)$. This language is denoted $L(A)$, and is defined by

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \text{ is in } F\}$$

That is, the language of A is the set of strings w that take the start state q_0 to one of the accepting states. If L is $L(A)$ for some DFA A, then we say L is a *regular language*

Nondeterministic Finite Automata

A "nondeterministic" finite automaton (*NFA*) has the power to be in serval states at once. This ability is often expressed as an ability to "guess" something about its input.

NFA's accept exactly the regular languages, just as DFA's do. However, NFA's are often more succinct and easier to design than DFA's. Moreover, while we can always convert an NFA to a DFA, the latter may have exponentially more staten than the NFA.

An Informal View of Nondeterministic Finite Automata

An NFA, like a DFA has a finite set of states, input symbols, a start state and a set of accepting states. It also has a transition function referred to by δ. However, in an NFA, δ can return a set of zero, one, or more states, rather than exactly one.

Definition of Nondeterministic Finite Automata

An NFA is represented essentially like a DFA:

$$A = (Q, \Sigma, \delta, q_0 m F)$$

where:

1. Q is a finite set of *states*.
2. Σ is a finite set of *input symbols*.
3. q_0, a member of Q, is the *start state*.
4. F, a subset of Q, is the set of *final* (or *accepting*) states.
5. δ, the *transition function* is a function that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q.

The Extended Transition Function

We need to extend the transition function δ of an NFA to a function $\hat{\delta}$ that takes a state q and a string of input symbols w, and returns the set of states that the NFA is in if it starts in state q and processes the string w.

The Language of an NFA

An NFA accepts a string w if it is possible to make any sequence of choices of next state, while reading the characters of w, and go from the start state to any accepting state. The fact that other choices using the input symbols of w lead to a nonaccepting state, or do not lead to any state at all (i.e., the sequence of states "dies"), does not prevent w from being accepted by the NFA as a whole. Formally, if $A = (Q, \Sigma, \delta, q_0, F)$ is an NFA, then

$$L(A) = \{ w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset \}$$

That is, $L(A)$ is the set of string w in Σ^* such that $\hat{\delta}(q_0, w)$ contains at least one accepting state.

Equivalence of Deterministic and Nondeterministic Finite Automata

DFA's and NFA's in practice have about the same number of states, although the DFA will will often have more transitions. In the worst case, however, the smallest DFA can have 2^n states while the smallest NFA for the same language has only n states.

The proof that DFA's can do whatever NFA's can do involves an important "construction" called the *subset construction* because it involves constructing all subsets of the set of state of the NFA.

The subset construction starts from an NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. ITs goal is the description of a DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L(D) = L(N)$. The other components of D are constructed as follows:

- Q_D is teh set of subsets of Q_N; i.e., Q_D is the *power set* of Q_N. Note that if Q_N has n states, then Q_D will have 2^n states. Often, not all these states are accessible from the start state of Q_D. Inaccessible states can be "thrown away," so effectively, the number of states of D may be much smaller than 2^n.
- F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \emptyset$. That is, F_D is all sets of N's states that include at least one accepting state of N.
- For each set $S \subseteq Q_N$ and for each input symbol a in Σ,

$$\delta_D(S, a) = \bigcup_{p \text{ in } S} \delta_N(p, a)$$

  That is, to compute $\delta_D(S, a)$ we look at all the states p in S, see what states N goes to from p on input a, and take the union of all those states.

Theorem: If $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ is the DFA constructed from NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ by the subset construction, then $L(D) = L(N)$.

Theorem: A language L is accepted by some DFA if and only if L is accepted by some NFA.

The Pigeonhole Principle

Colloquially, if you have more pigeons than pigeonholes, and each pigeon files into some pigeonhole, then there must be at least one hole that has more than one pigeon.

The pigeonhole principle may appear obvious, but it actually depends on the number of pigeonholes being finite. Thus, it works for finite-state automata, with the states as pigeonholes, but does not apply other kinds of automata that have an infinite number of states.

Dead States and DFA's Missing Some Transitions

We have formally defined a DFA to have a transition from any state, on any input symbol, to exactly one state. However, sometimes, it is more convenient to design the DFA to "die" in situations where we know it is impossible for any extension of the input sequence to be accepted. Technically, this automaton is not a DFA, because it lacks transitions on most symbols from each of its states.

However, such an automaton is an NFA. If we use the subset construction to convert it to a DFA, the automaton looks almost the same, but it includes a *dead state*, that is, a nonaccepting state that goes to itself on every possible input symbol. The dead state corresponds to $\emptyset$, the empty set of states of the

automaton.

In general, we can add a dead state to any automaton that has *no more* than one transition for any state and input symbol. Then, add a transition to the dead state from each other state q, on all input symbols for which q has no other transition. The result will be a DFA in the strict sense.

Finite Automata With Epsilon-Transitions

We now allow a transition on ϵ, the empty string. In effect, an NFA is allowed to make a transition spontaneously, without receiving an input symbol. Like nondeterminism, this new capability does not expand the class of languages that can be accepted by finite automata, but it does give us some added "programming convenience".

The Formal Notation for an ϵ-NFA

We represent an ϵ-NFA exactly as we do an NFA, with one exception: the transition function must include information about transitions on ϵ. Formally, we represent an ϵ-NFA A by $A = (Q, \Sigma, \delta, q_0, F)$, where all components have their same interpretation as for an NFA, except that δ is now a function that takes as arguments:

1. A state in Q, and
2. A member of $\Sigma \cup \{\epsilon\}$, that is, either an input symbol, or the symbol ϵ.

Epsilon-Closures

Informally, we ϵ-close a state q by following all transitions out of q that are labelled ϵ. However, when we get to other states by following ϵ, we follow the ϵ-transitions out of those states, and so on, eventually finding every state that can be reached from q along any path whose arcs are all labeled ϵ.

Extended Transitions and Languages for ϵ-NFA's

Suppose that $E = (Q, \Sigma, \delta, q_0 m F)$ is an ϵ-NFA. We first define $\hat{\delta}$, the extended transition function, to reflect what happens on a sequence of inputs. The intent is that $\hat{\delta}(q, w)$ is the set of states that can be reached along a path whose labels, when concatenated, form the string w. As always, ϵ's along this path do not contribute to w.

The language of an ϵ-NFA $E = (Q, \Sigma, \delta, q_0, F)$ is defined as: $L(E) = \{ w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset$. That is, the language of E is the set of strings w that take the start state to at least on accepting state.

Eliminating ϵ-Transitions

Given any ϵ-NFA E, we can find a DFA D that accepts the same language as E. Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$. Then, the equivalent DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

is defined as follows:

1. Q_D is the set of subsets of Q_E.
2. q_D = ECLOSE(S); that is, we get the start state of D by closing the set consisting of only the start state of E.
3. F_D is those set of states that contain at least one accepting state of E. That is, $F_D = \{ S \mid S \text{ is in } Q_D \text{ and } S \cap F_E \neq \emptyset \}$.
4. $\delta_D(S, a)$ is computed, for all a in Σ and sets S in Q_D by:
   (a) Let $S = \{p_1, p_2, \ldots, p_k\}$.
   (b) Computer $\bigcup_{i=1}^{k} \delta_E(p_i, a)$; let this set be $\{r_1, r_2, \ldots, r_m\}$.
   (c) Then $\delta_D(S, a) =$ ECLOSE($\{r_1, r_2, \ldots, r_m\}$).

Theorem: A language L is accepted by some ϵ-NFA if and only if L is accepted by some DFA.

Summary of Chapter 2

Deterministic Finite Automata A DFA has a finite set of states and a finite set of input symbols. One state is designated the start state, and zero or more states are accepting states. A transition function determines how the state changes each time an input symbol is processed.

Transition Diagrams It is convenient to represent automata by a graph in which the nodes are the states, and arcs are labeled by input symbols, indicating the transitions of that automaton. The start state is designated by an arrow, and the accepting states by double circles.

Language of an Automaton The automaton accepts strings. A string is accepted if, starting in the start state, the transitions caused by processing the symbols of that string one-at-a-time lead to an accepting state. In terms of the transition diagram, a string is accepted if it is the label of a path from the start state to some accepting state.

Nondeterministic Finite Automata The NFA differs from the DFA in that the NFA can have any number of transitions (including zero) to next states from a given state on a given input symbol.

The Subset Construction By treating sets of states of an NFA as states of a DFA, it is possible to convert any NFA to a DFA that accepts the same language.

ϵ-Transitions We can extend the NFA by allowing transitions on an empty input, i.e., no input symbol at all. These extended NFA's can be converted to DFA's accepting the same language.