Chapter 1.1: Finite Automata

5-tuple representation

We can describe a finite automaton M_1 formally with a 5-tuple by writing $M_1 = (Q, \Sigma, \delta, q_0, F)$ where:

- Q is the set of all **states** in the automaton
- Σ is the (finite) set of inputs readable by the automaton, called the **alphabet**
- $\delta : Q \times \Sigma \to Q$ is the **transition function**, which takes the two arguments of the current state (in Q) and the next input (in Σ), which together are in $Q \times \Sigma$, and returns the next state (in Q)
- q_0 is the **start state** ($q_0 \in Q$)
- $F \subseteq Q$ is the **set of accept states**; should the finite automata reach the end of its input string, the output is *accept*, otherwise it is *reject*. These states are sometimes called "final states".

Input strings and languages

To use a finite automaton, we give it a string of inputs $i_1, i_2, i_3, ...$ of length ℓ where each $i_n \in \Sigma \ \forall n \in [\ell] \subseteq \mathbb{N}$. In other words, we give the finite automaton a string of inputs, each of which is in the set of readable inputs Σ.

There are certain strings which finite automata may accept out of those given to it. The set of all such strings, which result in the *accept* signal, is called A. The set of strings accepted by finite automaton M is called the **language** of M, denoted $L(M) = A$. As A is a language and not a string, we say that M *recognizes* A (instead of M accepts A).

The empty language is denoted $\emptyset$, and the empty string is denoted ε. If a machine accepts no strings, then it recognizes the language $\emptyset$.

Factoid

Every automaton M accepts exactly one language, $L(M)$.

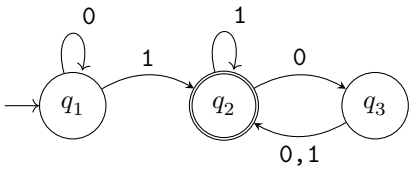
Each language A can be written in set notation (as each language is a set of strings) as follows:

$$A = \{w | P(w)\},$$

where $P(w)$ is some property of w, e.g. "w contains at least one 1 and an even number of 0s follow from the last 1".

State diagram representation

Small finite automata can also be represented informally using a state diagram, which is easier to grasp intuitively. An example, M_1, follows:



This diagram, while cumbersome to draw using `tikz` in LaTeX, is easier to visualize and can easily be transformed into the 5-tuple representation given earlier. This finite automaton M_1 is defined as

$$\begin{aligned} M_1 = (& \\ & Q = \{q_1, q_2, q_3\}, \\ & \Sigma = \{0, 1\}, \\ & \delta = \text{(defined below)}, \\ & q_0 = q_1, \\ & F = \{q_2\} \\ ), & \end{aligned}$$

where $\delta : Q \times \Sigma \to Q$ is defined as follows:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

Aside: NOT required course material for CSCI 320

This table is probably the best way to display the transition function. Those familiar with Markov chains (which can also be represented with state diagrams) might recall a transition matrix with probabilities; if the inputs can be probalistically determined, a transition matrix could likely be made that would represent δ. Consider the case in which a `0` had a 60% chance of appearing as the next letter of a string, and a `1` had a 40% chance of appearing. Then, we might represent the above automata as follows in Markov chain syntax:

$$\begin{bmatrix} 0.6 & 0.4 & 0 \\ 0 & 0.4 & 0.6 \\ 0 & 1 & 0 \end{bmatrix}$$

Note that the diagram in question has a closed communicating class $\{q_2, q_3\}$ and is thus not irreducible. We could calculate the stationary distributions of this Markov chain using this information, but as it is not very relevant, we will omit it.

Determining a finite automaton's acceptance criteria

In order to figure out what strings a finite automaton accepts or rejects, it is generally a good idea to create some short strings from the alphabet Σ and see how the automaton processes them. Going through this process for M_1 above will show you that $L(M_1) = \{w|P(w)\}$, where $P(w)$ is the same as the example property given previously.

However, this guess-and-check method is only really usable for small finite automata. With larger automata, or those which are defined formally rather than informally with a state diagram, it can be useful to think about the exact criteria for acceptance mathematically.

We say that finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ accepts a string $w = w_1, w_2, w_3, \cdots, w_n$ if and only if there exists a sequence of states $r_0, r_1, r_2, \cdots, r_n \in Q$ which satisfies the following three conditions:

- $r_0 = q_0$
- $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i \in \{0, 1, \cdots, n-1\}$
- $r_n \in F$

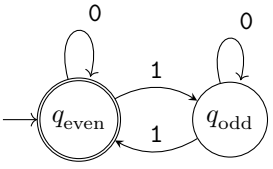
In other words: we say M accepts w if and only if there is a sequence of states that begins with the starting state, uses the transition function thereafter to transition between states, and ends in an accept state. With this, we say that M recognizes language A if $A = \{w|M \text{ accepts } w\}$.

Factoid

A language is called a **regular language** if some finite automaton recognizes it.

Designing finite automata

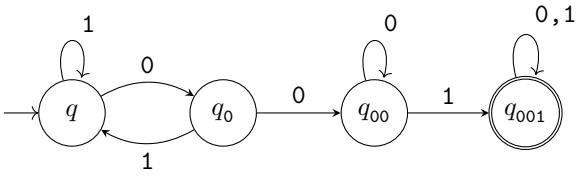
The suggested way to design a finite automaton around a language that you want it to recognize is to envision yourself as the automaton itself and think about what actions you would have to take in order to accept strings from that language and it alone. For example, consider the finite automaton M with alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$ and language $L(M) = \{w|P(w)\}$, and $P(w)$ is the proposition that w contains an even number of `1`s. Then, you (as the automaton) should be able to determine using your state alone whether, while reading the input string, you have read an even or odd number of `1`s at any given point. As such, it would make sense to have states for an even number and an odd number of `1`s. Then, as you start with zero `1`s with the empty string ε, you should start at the "even" state, and thereafter change states back and forth if and only if the next input in the string is a `1`. We might draw this out as follows:



When trying to accept only those strings which contain a certain *substring*, it is key to assign states to each possible substring of the given substring. Consider, as in the textbook example, the substring `001`, for some finite automaton with alphabet $\Sigma = \{0, 1\}$. In our automaton, we would need to assign states to account for the possible states of having read (consecutively) no symbols of the pattern, one `0`, two `0`s, and the entire substring `001`. We might assign these states the names q, $q_{\texttt{0}}$, $q_{\texttt{00}}$, and $q_{\texttt{001}}$.

When we are at q (having read no symbols in the substring), a `1` is meaningless to us (insofar as it does not allow us to make any progress along the substring); as such, we should simply stay put at q. If, however, we receive as our next input a `0`, we should proceed to $q_{\texttt{0}}$; then, if we receive a 1 we should return to q as we have created an incomplete substring. Should we receive another `0`, we should proceed to $q_{\texttt{00}}$. At this state comes the interesting conclusion that we are in the equivalent of a Markov chain's closed communicating class; that is, we will never go back to the previous states q and $q_{\texttt{0}}$. This is because an input of `0` will result in our remaining at the `00` portion of the substring, and an input of `1` will result in our going to $q_{\texttt{001}}$, after which we know that we have found the required substring. At this last state, we will simply stay put regardless of the next input, as we know that we have found the substring for which we were looking. This last state is also the accept state.

This might be difficult to understand when just reading, so a diagram is included below for convenience's sake. Note that it might be useful to draw out such diagrams when thinking about nontrivial automata such as the one described above.



Regular operations

We define **regular operations** as the three different operations on languages, which are as follows:

- **Union**: $A \cup B = \{x|x \in A \vee x \in B\}$
- **Concatenation**: $A \circ B = \{xy|x \in A \wedge x \in B\}$
- **Star**: $A^* = \{x_1 x_2 x_3 \ldots x_k | k \geq 0,\ x_i \in A\ \forall i\}$

The first of these operations is familiar – the union operation ($\cup$) is essentially the same as the one for sets. The other two, however, are new: the concatenation operation ($\circ$) might seem similar to the "and" ($\wedge$) operator seen in propositional logic, but in reality is different as we are now working with strings rather than propositions. It is more similar to the concatenation of strings in most programming languages, e.g. `A + B`, which entails putting the characters in the first string `A` before those in `B`. The major difference lies in the fact that we

are here concerned with *all possible outputs* in which (to continue with our analogy) characters from string `A` come before characters from string `B`. An example from the textbook follows, in which A and B are sets of strings rather than strings themselves:

Consider sets A and B having contents {`good, bad`} and {`boy, girl`} respectively. Then:

$$A \cup B = \{\texttt{good, bad, boy, girl}\}$$
$$A \circ B = \{\texttt{goodboy, goodgirl, badboy, badgirl}\}$$

Back to our definitions, the **star** operation as defined above is unique among the regular operations in that it refers to only one language (and is thus a *unary* operator rather than a *binary* one). This operation is similar (but not identical) to the *power set* operation $\mathcal{P}$ on sets, which returns the set of all subsets of a given set. For example, $\mathcal{P}(\{1,2\}) = \{\emptyset, \{1\}, \{2\}, \{1,2\}\}$, where $\emptyset$ (sometimes also written $\{\}$) is the empty set (the set containing zero elements). The star (*) operator is similar in that it contains the "regular language-equivalent" of the empty set, which is the empty string ε, and also contains the individual elements in the language as well as collections of them; however, it differs in that the elements of a star operation are not sets themselves but rather strings, and are thus not prevented from containing duplicate elements within themselves, or permutations of already-existing strings in the set. To illustrate this, the above example from the textbook is continued:

$$A^* = \{\varepsilon, \texttt{good}, \texttt{bad}, \texttt{goodgood}, \texttt{goodbad}, \texttt{badgood}, \texttt{badbad}, \texttt{goodgoodgood}, \texttt{goodgoodbad}, \texttt{goodbadgood}, \texttt{goodbadbad}, \dots\}$$

Note that while A^* is an infinite set, $A \cup B$ and $A \circ B$ are finite.

Now we move onto the notion of **closure**. We say that a collection of objects is *closed* under an operation if applying that operation to members of the collection returns an object still in the collection. This is different from the notion of the closure of a set described in topology, in which a set contains its limit points, and is rather more similar to the notion of closure described in the group theory of abstract algebra.

We use the above-described notion of closure to describe the behaviour of a regular expression under the various regular operations – namely, we have the following Factoid:

Factoid

The class of regular languages is closed under both the *union* ($\cup$) and *concatenation* ($\circ$) operations.

The proof for these follows.

First, we want to prove that the class of regular languages is closed under the $\cup$ operation – that is, the union of two regular languages is itself a regular language. Consider regular languages A_1 and A_2, and their union $A_1 \cup A_2$. We know that some finite automatons recognize A_1 and A_2, and we want to prove that some finite automaton recognizes $A_1 \cup A_2$. In other words, for machines M_1 and M_2 associated with A_1 and A_2 respectively, we want the machine M associated with $A_1 \cup A_2$ to accept the same inputs as both of A_1 and A_2.

The way we can do this is by considering every possible ordered pair of states, one from M_1 and one from M_2. To this end, we want to take the Cartesian product $Q_{M_1} \times Q_{M_2}$ of the sets of states Q_{M_1} and Q_{M_2}, whose size will be (as we learned in lecture) $|Q_{M_1}| \cdot |Q_{M_2}|$. Then, we have a state in M for every single one of these possible pairs, accounting for both M_1 and M_2 at the same time. The automaton M will transition to the relevant pairs as dictated by some combination of the transition functions δ_{M_1} and δ_{M_2}, and will have accept states when *either* of the states M_1 or M_2 in the ordered pair are an accept state.

More formally, we prove the above as follows (from the textbook):

Let M_1 recognize A_1, where $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and M_2 recognize A_2, where $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. We will prove the above by construction.

Construct M to recognize $A_1 \cup A_2$, where $M = (Q, \Sigma, \delta, q_0, F)$.

- $Q = \{(r_1, r_2) | r_1 \in Q_1 \wedge r_2 \in Q_2\}$. This set is the Cartesian product of sets Q_1 and Q_2 as outlined above, written $Q_1 \times Q_2$.
- Σ, the alphabet, is the same as in M_1 and M_2. We assume for simplicity that M_1 and M_2 have the same alphabet. The theorem remains true even for different alphabets, for which we would modify Σ to be $\Sigma_1 \cup \Sigma_2$.
- δ, the transition function, is defined formally as follows. For each $(r_1, r_2) \in Q$ and each $a \in \Sigma$, let
$$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)).$$
By combining the transition functions in this manner, we get the next state for each of M_1 and M_2 as an ordered pair, which is (by definition) in M's state space Q and therefore valid.
- q_0 is the pair of inital states of M_1 and M_2, (q_1, q_2).
- F is the set of pairs in which either member is an accept state of M_1 or M_2. We can write it more formally as
$$F = \{(r_1, r_2) | r_1 \in F_1 \vee r_2 \in F_2\},$$
or alternatively as
$$F = (F_1 \times Q_2) \cup (F_2 \times Q_1).$$
Note that it is *NOT* the same as $F = F_1 \times F_2$, which would mean that M would only accept an input string if *both* of the states in the pair were accept states. This would be the case if we wanted the intersection ($\cap$)

rather than the union ($\cup$), and actually serves as a convenient corollary that the class of regular languages is also closed under intersection.

To prove that this is necessarily a *correct* construction, we might proceed to use mathematical induction. It is omitted here, as it is omitted in the textbook, for brevity. (Also, it shouldn't come up on any exams.)

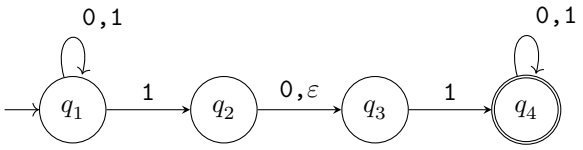
Now that we have proven that the class of regular languages is closed under union, we want to prove that it is closed under concatenation. Consider regular languages A_1 and A_2, and their concatenation $A_1 \circ A_2$. The issue here is that we need to construct some automaton M that accepts inputs broken into two pieces, in which M_1 accepts the first piece and M_2 accepts the second piece. In order to deal with this we need something called **nondeterminism**, which will be covered in the second unit of Chapter 1.

Chapter 1.2: Nondeterminism

Nondeterministic finite automata

So far we've been working with *deterministic* finite automata, or DFAs. Now we'll look at *nondeterministic* finite automata, or NFAs. The major difference between DFAs and NFAs is that NFAs don't have a deterministic outcome based on a single traversal of states. Rather than having a transition function which explicitly defines which single state will be moved to given a current state and an input, as in DFAs, NFAs can have multiple possible states that are reached as a result of a given input. As such, the criteria for acceptance is the reaching of any accept state by the end of the evaluation of the string by *any of the possible paths taken by the automaton*.

Consider the following diagram of an NFA:



Note that there are two different paths that the automaton can take from q_1 given an input of `1`: it can stay at q_1, or it can move to q_2. This generates two different paths – one which considers the rest of the input as though the current state is q_1 and one which considers the rest of the input as though the current state is q_2. These paths will traverse the rest of the NFA similarly.

Also, note the lack of a path from q_3 given an input of `0`, and the lack of a path from q_2 given an input of `1`. If either of these inputs without a path are encountered by the NFA while at the given states, the path "dies" and is rejected. If all the paths generated from an input are either rejected (or "killed") or fail to reach an accept state by the time the input string is finished processing, then, similarly to a DFA, the NFA will reject the input string. In fact:

Factoid

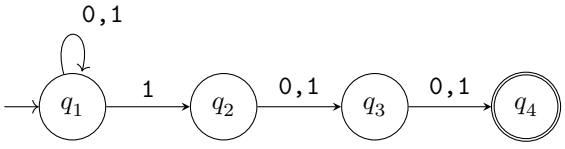
Every DFA is an NFA. NFAs are a more generalized version of DFAs, which may have additional features.

One of these additional features hasn't been mentioned yet, which is the ε above the arrow between q_2 and q_3. This ε, which refers to the empty string as has been reiterated many times, means that one of the paths that can be taken by the NFA upon reaching q_2 is to immediately head to q_3, without the need for an input character. As such, inputs may travel from q_1 directly to q_3 when one input has been read in this NFA, despite the fact that only one "step" has been taken.

When conceptualizing the way an NFA processes input strings, it can be useful to think of the processing of the input string as creating *forking* processes whenever multiple paths can be taken. Ideas of parallel computing, threads, and concurrency can be used as analogies here. Another useful analogy is that of an (upside-down) tree, in which each intersection of multiple paths given the current input creates multiple branches. If any of the "leaves" of the tree at the end of its evaluation of the input string are at an accept state, the entire tree is greenlighted and the input string is accepted; otherwise it is rejected. If none of the leaves are at an accept state, then, similarly to the DFA, the tree (input string) is rejected.

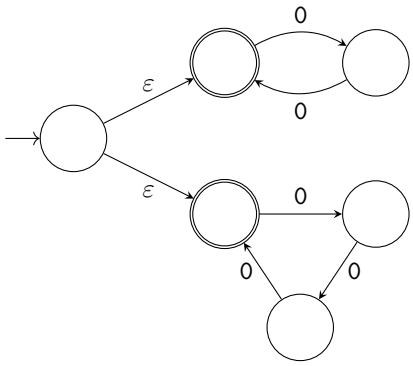
Plugging in various strings from the usual alphabet $\Sigma = \{\texttt{0},\texttt{1}\}$ will lead the reader to find that the above example of an NFA will accept any input string that contains either `101` or `11` as a substring. The check for this is left as an exercise to those reading the notes.

Another more intuitive example is as follows. Consider an NFA which accepts only those strings that contain a `1` as the third-to-last input on an input string. This could be represented as follows:



Every NFA can be converted into a corresponding DFA, but it isn't always easy – in fact, it's frequently very difficult. The smallest DFA for this intuitive example, for instance, has no fewer than eight states, and is significantly more difficult to comprehend at a glance than the equivalent NFA. (The depiction of said DFA is omitted here – curious readers can check the textbook.)

Another (slightly more complex) example follows. Consider the following NFA:



This NFA, which has the *unary alphabet* $\Sigma = \{0\}$, illustrates the usefulness of the ε transition. In this case, the εs are able to arbitarily split the NFA's tree into two separate paths into two closed communicating classes, which accept input strings of zeroes of a length that is a multiple of either 2 or 3.

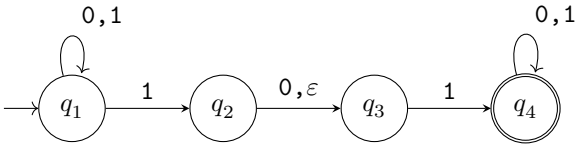
Formal definition of nondeterministic finite automata

Now that we've gotten a good idea of what NFAs are, it's time to define them formally. The definition of NFAs is exceedingly similar to that of DFAs, which should make sense given both that it is possible to convert between them and that NFAs are a generalization of DFAs.

The definition is as follows:

A **nondeterministic** finite automaton is, like a DFA, defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. Each of these have their usual meanings, except for the transition function δ. In a NFA, the transition function is (instead of a function $\delta : Q \times \Sigma \to Q$) a function $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$. We define Σ_ε to be the union of the alphabet Σ and the empty string ε, and use our working definition of $\mathcal{P}(Q)$ as the power set of the set of states Q. In other words, δ represents a transition from the ordered pair of the current state and the next input letter *or* ε to some subset of the set of states. Each state-letter combination might yield a different state, or the same state-letter combination might yield multiple states, and all of these possiblities are accounted for by the power set.

We redraw our first NFA here:



This NFA, N_1, has the following 5-tuple representation:

$$
\begin{aligned}
N_1 = (\\
&Q = \{q_1, q_2, q_3, q_4\},\\
&\Sigma = \{\texttt{0},\texttt{1}\},\\
&\delta = \text{(defined below)},\\
&q_0 = q_1,\\
&F = \{q_4\}\\
),
\end{aligned}
$$

where $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is defined as follows:

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

Computation for an NFA is defined similarly to that for a DFA. For NFA $N = (Q, \Sigma, \delta, q_0, F)$, and string w over alphabet Σ, we say N accepts w if we can write w as $y_1 y_2 \cdots y_m$, where each y_1 is in Σ_ε and a sequence of states $r = r_0, r_1, \ldots, r_m$ exists in Q where $r_0 = q_0$ the initial state of N, $r_{i+1} \in \delta(r_i, y_{i+1})$ $\forall i \in \{0, 1, \ldots, m-1\}$, and $r_m \in F$. In other words, r starts at the starting state, ends in an accept state, and follows the transition function δ in between those two states. The difference here between DFAs and this NFA definition is that r_{i+1} is *in the set* $\delta(r_i, y_{i+1})$ rather than being equal to the next transition, as the return value of a transition function in an NFA is always a set in $\mathcal{P}(Q)$.

Equivalence of NFAs and DFAs