Languages, regular languages, finite automata

Content largely taken from Richards [1] and Sipser [2]

1 Languages

An alphabet is a finite set of characters, which we will often denote by $\Sigma$. For example, $\Sigma = \{0, 1\}$ is an alphabet that we will frequently use. A language over some alphabet $\Sigma$ is a set of strings made up of characters from $\Sigma$. For example, $L_1 = \{0, 1, 00, 11\}$ is a language over the alphabet $\Sigma = \{0, 1\}$. An English dictionary is also a language, over the alphabet $\{a, \ldots, z, A, \ldots, Z\}$. However, languages need not be finite size: “the set of all binary strings ending in 0” is a language over $\Sigma = \{0, 1\}$. Clearly such a language is not as easy to formally describe, but we will address that issue later on.

It is useful to define a few set operators for languages. The union operator, $\cup$, is defined as

$\forall L_1, L_2, \exists \cup L_1 \cup L_2 = \{x \mid x \in L_1 \vee x \in L_2\}$.

For any language $L$, we define $L^0 = \{\Lambda\}$, where $\Lambda$ is a special string, called the empty string. For $k > 0$, we recursively define $L^k = LL^{k-1}$. That is, we concatenate $L$ with itself $k$ times (and also include the empty string).

Finally, we define the closure operator: $L^* = \bigcup_{i=0}^{\infty} L^i$.

At a high level, the fundamental question of computer theory is the following. Given a language $L$ and some string $x$, how hard is it to determine whether $x \in L$? (Or, as we will often phrase this question, how hard it is to decide the language $L$?) We all have some intuition of what this means. For example, given a string of English characters, one can determine whether it is a valid English word by scanning through the English dictionary, one word at a time. But anyone that still uses a paper dictionary knows that they can do better using binary search, and we may even have seen a proof that you require $O(\log n)$ comparison for a dictionary of size $n$. Such algorithmic questions aren’t really our focus in this class, though. Rather, we are interested in characterizing whole classes of languages. A language that can be decided in polynomial time on a Turing machine is said to be in a class of languages that we call $\mathcal{P}$. But are there languages that are fundamentally easier to decide than these? Are there languages that can be proven to be strictly harder than those in $\mathcal{P}$? And, furthermore, why is the Turing machine the right model for computation? What if we consider other models? And why is time the right metric: how do memory and communication constraints impact what we can compute? Does access to good random sources help us to decide more languages? Moreover, why is deciding whether some string $x \in L$ the right place to focus our attention? We will look at almost all of these questions, and more, with the aim of gaining a deeper fundamental understanding of what is possible in our field and what is not.
Finite Automata

2.1 Regular Languages

To begin, we start with a very simple model of computation, and a simple class of languages, which are called the regular languages. The regular languages correspond to those generated by regular expressions. We formalize this class of languages recursively, as follows. $\mathcal{R}$ will denote the set of all regular languages over some alphabet, $\Sigma$.

1. $\emptyset \in \mathcal{R}$ and $\{\Lambda\} \in \mathcal{R}$.
2. $\forall \sigma \in \Sigma : \{\sigma\} \in \mathcal{R}$.
3. If $L \in \mathcal{R}$ then $L^* \in \mathcal{R}$.
4. If $L_1 \in \mathcal{R}$ and $L_2 \in \mathcal{R}$ then $L_1L_2 \in \mathcal{R}$.
5. If $L_1 \in \mathcal{R}$ and $L_2 \in \mathcal{R}$ then $L_1 \cup L_2 \in \mathcal{R}$.

These languages are so common and useful, that we frequently use a special notation, called regular expressions in order to specify languages in this class. In this notation, the ‘{’ and ‘}’ are dropped, and union is denoted by ‘+’. For example, $(ab + c)^* = \{\Lambda, ab, c, abc, cc, cab, abab, \ldots\}$

2.2 Deterministic Finite Automata

A deterministic finite automata is a state machine that takes an input string and either accepts or rejects that string. It makes this decision by transitioning through a sequence of states, making exactly one transition for each character of the input string in a deterministic way. After the transitions are complete, it accepts if it has terminated in a state marked “accept”, and it rejects if it has stopped in a state marked “reject”. We will formalize this model of computing in a moment, but it is helpful to first demonstrate it by example. In the first example in Figure 1, there is a special start state, labeled ‘A’, and the state labeled ‘C’ has a circle around it, denoting that it is an accept state. We note that there can be multiple accept states, though that isn’t the case in these two examples. Consider input string ‘bc’: the DFA reads the first character, and transitions from state ‘A’ to state ‘B’. It then reads the second character and transitions into the accept state, ‘C’. Because this is the last character of input, the machine terminates in the accept state, and we say that the machine accepts input ‘bc’. Equivalently, we say that ‘bc’ is in the language of this DFA. Consider now input ‘bcc’: the last character causes a transition out of the accept state and into state ‘D’, where the machine now terminates. Because ‘D’ is not marked as an accept state, we say that the DFA rejects this input, or that the input is not in the language of this DFA. Looking more closely at state ‘D’, you can see that it is a sink: there are no transitions out of ‘D’, so no input that leads to state ‘D’ will ever be accepted. This is sometimes called a trap state, and usually we will leave such states out of our diagrams in order to simplify them. Removing ‘D’ and all of its edges, we will sometimes find that while processing an input string, there is no transition that can be made. In this case, we interpret this though we had transitioned to a trap state, and say that the machine rejects the input.

The language of the second DFA in Figure 1 is the language of the regular expression $(a + bb)^*$. (Written as a regular language, this would be $L^*$, where $L = \{a\} \cup \{bb\}$.) Note that $\Lambda$ is in this language, by the definition of the closure operator; to see why $\Lambda$ is accepted by the DFA, note
that the start state is also an accept state. We will shortly show that all regular languages can be decided by DFAs. Actually, we will show more: that the class of regular languages is equivalent to the class of languages decided by DFAs.

2.3 Formally Defining DFAs

Formally, a DFA is defined by an alphabet $\Sigma$, a finite set of states, $Q$, a start state, $S \in Q$, a set of accept states, $A \subseteq Q$, and a transition function $\delta : Q \times \Sigma \rightarrow Q$. Putting this together, $M = (\Sigma, Q, S, A, \delta)$. Returning to the first example in Figure 1 (and ignoring the trap state), this DFA can be formally described by $(\Sigma = \{a, b, c\}, Q = \{A, B, C\}, A, A = \{C\}, \delta)$, where $\delta$ is defined as:

$$
\delta = \begin{array}{ccc}
A & a & b & c \\
C & B & \bot & \bot \\
B & \bot & \bot & C \\
C & \bot & \bot & \bot \\
\end{array}
$$

The formalism will help us to prove things about DFAs and the languages that they decide. Formally, for any alphabet $\Sigma$, and any $x \in \Sigma$, we can see that there exists a DFA deciding the language $\{x\}: (\Sigma, Q = \{A, B\}, A, B, \delta)$, where $\delta(A, x) = B$, and $\delta$ otherwise has output $\bot$. Informally, the DFA has a start state that is non-accepting, and a single accept state. The only transition allowed goes from the start to the accept state on input $x$. We can also define a DFA for $\{\Lambda\}$: it has a single state that is both the start state and an accept state, and it does not allow any transitions. So we can see that the languages defining the “base-cases” of the regular languages are all decidable by DFAs.

We now show that if a language $L_1$ is decided by DFA $M_1$, and a language $L_2$ is decided by DFA $M_2$, then there exists a DFA $M$ that decides $L_1 \cup L_2$. Intuitively, we construct $M$ so that it tracks the movement of the input through both $M_1$ and $M_2$, simultaneously. To do that, we create $|Q_1| \cdot |Q_2|$ states, and label each with a pair of names, one from $Q_1$ and one from $Q_2$. If $M$ is in state $(A, B)$, we can think of this as indicating that $M_1$ would currently, on this input, be in state $A$, while $M_2$ would currently be in state $B$. If $M$ halts in state $(A, B)$, we want to accept if either $A$ is an accept state for $M_1$, or if $B$ is an accept state for $M_2$.

To simplify the formal exposition, we’ll assume $M_1$ and $M_2$ share the same alphabet; it is easy to see that this isn’t necessary. Let $M_1 = (\Sigma, Q_1, S_1, A_1, \delta_1)$ and let $M_2 = (\Sigma, Q_2, S_2, A_2, \delta_2)$.
Then $M = (\Sigma, Q, S, A, \delta)$ is defined as follows. $Q = \{(A, B) \mid A \in Q_1 \land B \in Q_2\}$. $S = (S_1, S_2)$, $A = \{(A, B) \mid A \in A_1 \lor B \in A_2\}$, and $\delta((A, B), x) = (\delta_1(A, x), \delta_2(B, x))$.

To prove that $L(M) = L_1 \cup L_2$, we must show two things. First, we prove that if $w \in L(M)$, then $w \in L_1 \cup L_2$. We will write $w = w_1 \ldots w_k$, letting $w_i$ denote the $i$th character of $w$. Note that $w \in L(M)$ implies that there is a sequence of states in $M$, $(S_1, S_2), (A_1, B_1), (A_2, B_2), \ldots, (A_k, B_k)$ such that $\delta((S_1, S_2), w_1) = (A_1, B_1)$, $\delta((A_i, B_i), w_{i+1}) = (A_{i+1}, B_{i+1})$, and either $A_k \in A_1$, or $B_k \in A_2$. Without loss of generality, let’s assume that $A_k \in A_1$. It follows by the way $M$ was constructed that $\delta_1(S_1, w_1) = A_1$, and, for $i \in \{1, \ldots, k - 1\}$, $\delta_1(A_i, w_{i+1}) = A_{i+1}$. Since $A_k \in A_1$, it follows that $M_1$ accepts $w$, and that $w \in L_1 \cup L_2$. Secondly, we must show that if $w \in L_1 \cup L_2$, then $M$ accepts $w$. We leave this direction as an exercise. We will later come back to the other regular operators, closure and concatenation.

### 2.4 Non-deterministic Finite Automata (NFAs)

We consider a very useful relaxation in how we model finite automata. Although it was not made explicit, we previously did not allow any ambiguity in how our transitions were to be made: for any state $A$ and any input character $x$, we have, so far, allowed only a single transition from $A$ to be labeled with $x$. Relaxing that gives us a lot more flexibility in our design. Consider the two examples in Figure 2, again taken from Richards [1]. Both machines decide the same language: $\{w \in \{a, b\}^* \mid w \text{ ends in } ab\}$. The second example, which is non-deterministic, has an ambiguous transition out of the start state: on input ‘a’, the machine has a choice to make. It could either transition to state $q_1$, or it could stay in the start state. We say that this machine accepts an input if there exists some sequence of allowable transitions that ends in an accept state. Importantly, we only require the existence of some such sequence of transitions: we do not require that all allowable transitions result in acceptance, and we do not care how one might find such a sequence.

An even better example of where non-determinism helps ease the design of a finite automata is the following language. $L = \{x \in \{a, b\}^* \mid \text{the } k\text{th symbol from the last is } 'a'\}$, where $k$ is some fixed integer. We described an NFA for this language in class, and we will design a DFA for this language in the homework.

To formally define NFAs, we have to change the definition of our transition function. Whereas in DFAs, we have $\delta : Q \times \Sigma \to Q$, we now have to allow $\delta$ to map the same domain to a set of states, rather than to a single state. Formally, $\delta : Q \times \Sigma \to 2^Q$, where $2^Q$ denotes the power-set. Looking again at example 2 in Figure 2, we have

![Figure 2: An example of an NFA (taken from Richards [1])](image)
Additionally, it is helpful to allow \( \Lambda \) transitions. These transitions allow the machine to move from one state to another without using up any of the input string. We don’t bother to formalize this.

### 2.5 Equivalence of DFAs and NFAs

How much additional power does this non-determinism give us? It seems to make machine design a lot simpler, but does it allow us to decide a larger class of languages? It turns out that it does not: the set of languages decidable by NFAs is exactly the regular languages, just as for DFAs. We prove now that the two models are equivalent in this sense.

It is clear from the definitions that every DFA is also an NFA, so we only need to show that for every NFA, \( M = (\Sigma, Q, q_0, A, \delta) \), there exists a DFA, \( M' = (\Sigma, Q', S', \mathcal{A}', \delta') \), such that \( L(M') = L(M) \). The intuition is similar to the one above for showing a DFA that decides the union of two languages. We will create a new state for every possible subset of \( T \in Q \). We can think of a state \((q_1, q_5, q_7)\) as capturing the fact that \( M \) could have followed paths leading to state \( q_1, q_5, \) or \( q_7 \). In this way, our DFA will keep track of all the possible places we could currently be in the NFA, given the input string seen so far. With that intuition, the state \((q_1, q_5, q_7)\) is an accept state if any of \( q_1, q_5, \) or \( q_7 \) are accept states for \( M \). Formally, \( Q' = 2^Q, S' = \{q_0\}, \mathcal{A}' = \{T \in Q' \mid \exists t \in T \text{ s.t. } t \in A\}, \) and \( \delta'(T, x) = \bigcup_{q \in T} \delta(q, x) \).

We need to prove two things. For \( w = w_0 \cdots w_{k-1} \), if \( w \in L(M) \), then \( w \in L(M') \), and if \( w \in L(M') \), then \( w \in L(M) \). We start with the first statement, letting \( w \in L(M) \). Because \( M \) is an NFA, we know that there exists some sequence of states, \( q_0, q_1, \ldots, q_k \), such that, \( q_{i+1} \in \delta(q_i, w_i) \), and \( q_k \in A \). Let \( T_0, T_1, \ldots, T_k \), be the states in \( M' \) such that for each \( i \in \{0, \ldots, k\} \), \( T_{i+1} = \delta'(T_i, w_i) \). We claim that \( T_k \in \mathcal{A}' \). To show this, we first argue that \( q_i \in T_i \). This clearly holds for \( q_0 \), since \( T_0 = S' = \{q_0\} \). Assume it holds for \( q_i \), and recall that by the definition of \( \delta' \), \( T_{i+1} = \bigcup_{q \in T_i} \delta(q, w_i) \). Since \( q_{i+1} \in \delta(q_i, w_i) \), and \( q_i \in T_i \), it follows that \( q_{i+1} \in T_{i+1} \). Finally, since \( q_k \in T_k \), and \( q_k \in A \), it follows that \( T_k \in \mathcal{A}' \).

We now need to prove that if \( w \in L(M') \), then \( w \in L(M) \). Using the same notation, let \( T_0, \ldots, T_k \) be the sequence of states such that \( T_{i+1} = \delta'(T_i, w_i) \). We have to show that there exists some sequence of states in \( Q, q_0, \ldots, q_k \), such that \( q_{i+1} \in \delta(q_i, w_i) \), and \( q_k \in A \). We start by choosing \( q_k \), and work backwards. To choose \( q_k \), we note that because \( T_k \in \mathcal{A}' \), then by definition of \( \mathcal{A}' \), there is some \( q_k \in T_k \) such that \( q_k \in A \). Choose any such \( q_k \). For \( i < k \), assume \( q_{i+1} \) has already be chosen. Since \( T_{i+1} = \bigcup_{q \in T_i} \delta(q, w_i) \), there exists some \( q_i \in T_i \) such that \( q_{i+1} \in \delta(q_i, w_i) \). Choose any such \( q_i \), and repeat. Since \( T_0 = \{q_0\} \), we can (and must) choose \( q_0 \) as our start state. This concludes the proof.

Actually, technically, we also have to show how to handle \( \Lambda \)-transitions when constructing \( M' \). This is easily done. For each \( q \in Q \), let \( E(q) \) be the set of states that is reachable using only \( \Lambda \)-transitions. Then, instead of defining \( \delta'(T, x) = \bigcup_{q \in T} \delta(q, x) \), we define it as \( \delta'(T, x) = \bigcup_{q \in T} (\delta(q, x) \cup E(q)) \). The rest of the proof would proceed as before.
2.6 Equivalence of Regular Languages and DFAs

Using NFAs, it becomes much easier to show that all regular languages can be decided by a DFA. We leave it as a homework problem to show that

1. if a language $L$ is decidable by a DFA, then $L^*$ is decidable by some NFA, $M$.

2. if $L_1$ and $L_2$ are decided by DFAs $M_1$ and $M_2$, then $L = L_1 L_2$ is decidable by some NFA, $M$.

To complete the proof that the class of regular languages and the class of languages decidable by DFAs are the same, we must also show that every language that is decidable by a DFA is regular. This is not a difficult proof, but we omit it in this class so that we can move on to other interesting things.

References
