10 The Pumping Lemma for Regular Languages

We know now how to prove that a language *is* regular; we have at least two ways—exhibit a DFA and prove that it accepts the language or exhibit a regular expression and prove that it denotes it. We know also that every regular language can be accepted by a DFA, which is to say that there is a finite bound (independent of the length of the input) on the amount of memory needed to recognized the strings in the language. But this is not a very powerful model. It seems likely that there are languages that cannot be recognized using a finitely bounded amount of memory. One example, from the informal examples we explored at the start of the tutorial, is the language

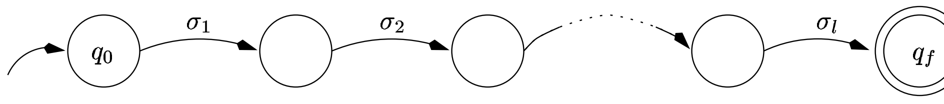
$$L_{ab} = \{a^i b^i \mid i \geq 0\}.$$

Here our intuition was that we can put no bound on number of 'a's we have to count in order to recognize the language. Consequently, it appears that this is not a regular language. The question is how to prove that a language is *not* regular.

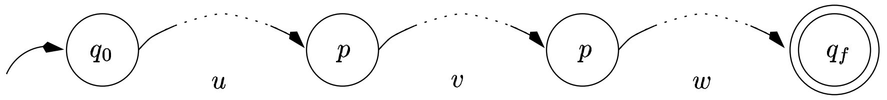
Potentially, this is a much more difficult problem. To establish that a language is regular we need only exhibit a DFA that accepts it; to establish it is not regular we need to prove that it is not accepted by *any* DFA. Fortunately, by looking at the nature of DFAs we can identify properties that are characteristic of the languages they accept. Thus, the fact that regular languages are all accepted by DFAs implies that they share these properties. The approach we will use to prove non-regularity of a language is to show that it does not share at least one of these properties. Consequently, it cannot be accepted by a DFA. In this section, we will explore a closure property of the following sort: if a regular language includes strings of a certain type, then it includes all strings of a related type. We will be able to establish non-regularity of a given language, then, by exhibiting a string of the first type that is in the language along with a string of the related type that is not.

Lets start by considering what it means for a string to be accepted by a DFA. Suppose $\mathcal{A} = \langle Q, \Sigma, q_0, \delta, F\rangle$ is a DFA and that $Q = \{q_0, q_1, \ldots, q_{n-1}\}$ includes n states. Thinking of the automaton in terms of its transition graph, a string x is accepted by the automaton iff there is a path through the graph from q_0 to some $q_f \in F$ that is labeled x, i.e., if $\hat{\delta}(q_0, x) \in F$. Suppose $x \in L(\mathcal{A})$ and $|x| = l$. Then there is a path l edges long from q_0 to q_f. Since the path traverses l edges, it must visit $l + 1$ states.

$$x = \sigma_1\sigma_2\cdots\sigma_l$$



Suppose, now, that $l \geq n$. Then the path must visit at least $n+1$ states. But there are only n states in Q; thus, the path must visit at least one state at least twice. (This is an application of the *pigeon hole principle*: If one places k objects into n bins, where $k > n$, then at least one bin must contain at least two objects.)



Thus, whenever $|x| \geq n$ the path labeled w will have a cycle. We can break the path into three segments: $x = uvw$, where

- there is a path (perhaps empty) from q_0 to p labeled u (i.e., $\hat{\delta}(q_0, u) = p$),
- there is a (non-empty) path from p to p (a cycle) labeled v (i.e., $\hat{\delta}(p, v) = p$),
- there is a path (again, possibly empty) from p to q_f labeled w (i.e., $\hat{\delta}(p, w) = q_f$).

But if there is a path from q_0 to p labeled u and one from p to q_f labeled w then there is a path from q_0 to q_f labeled uw in which we do not take the loop labeled v, which is to say $uw \in L(\mathcal{A})$. Formally

$$\hat{\delta}(q_0, uw) = \hat{\delta}(\hat{\delta}(q_0, u), w) = \hat{\delta}(p, w) = q_f \in F.$$

Similarly, we can take the v loop more than once:

$$\begin{aligned}
\hat{\delta}(q_0, uvvw) &= \hat{\delta}(\hat{\delta}(\hat{\delta}(\hat{\delta}(q_0, u), v), v), w) \\
&= \hat{\delta}(\hat{\delta}(\hat{\delta}(p, v), v), w) \\
&= \hat{\delta}(\hat{\delta}(p, v), w) \\
&= \hat{\delta}(p, w) = q_f \in F.
\end{aligned}$$

In fact, we can take it as many times as we like. Thus, $uv^iw \in L(\mathcal{A})$ for all i.

This implies, then, that if the language accepted by a DFA with n states includes a string of length at least n then it contains infinitely many closely related strings as well. We can strengthen this by noting (as a consequence of the pigeon hole principle again) that the length of the path from q_0 to the first time a state repeats (i.e., the second occurrence of p) must be no greater than n. Thus $|uv| \leq n$.

Now suppose L is an arbitrary regular language. Then there is *some* DFA accepting it and that DFA has *some* fixed number of states. Thus there is a constant n (the number of states in a DFA accepting L) such that if L includes any string of length greater than or equal to n then there is a non-empty segment of that string falling somewhere in its first n positions that can be repeated (or *pumped*) any number of times (including zero) always producing strings in L. This result is known as the *Pumping Lemma* for regular languages.

Lemma 13 (Pumping Lemma) *For every regular language L there is a constant n depending only on L such that, for all strings $x \in L$ if $|x| \geq n$ then there are strings u, v and w such that*

1. $x = uvw$,
2. $|uv| \leq n$,
3. $|v| \geq 1$,
4. *for all* $i \geq 0, \quad uv^iw \in L$.

What this says is that if there is any string in L "long enough" then there is *some* family of strings of related from that are all in L, that is, that there is *some* way of breaking the string into segments uvw for which uv^iw is in L for all i. It *does not* say that *every* family of strings of related form is in L, that uv^iw will be in L for every way of breaking the string into three segments uvw. It also *does not* say that every long string in L is of form uv^iw for some $i > 1$ (i.e., it *does not* say that every long string in L has some part that repeats). It does not, in fact, say anything about individual strings at all, it simply lets us identify families of strings all of which must be in L. The way we will use this is to identify such a family that should be in L if L is regular for which we can show at least one string in the family is not in L.

Note that this lemma talks *only* about the strings in the language. While we justified it by appealing to the fact that a language is regular iff there is a

DFA that accepts it, we don't need to actually come up with such a DFA to apply it. On the other hand, if we actually *have* a DFA for the language we can fix a concrete upper bound on the constant of the lemma—n is no larger than the number of states in Q. Thus we can strengthen the pumping lemma slightly, by adding:

Lemma 14 *Suppose $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ is a DFA witnessing the fact that L is regular. Then the constant n of the pumping lemma is no greater than* $\mathbf{card}(Q)$.

10.1 Applying the Pumping Lemma

To establish that a language L is not regular using the pumping lemma we need to show that the pumping lemma is not true for that language, i.e., for *any* n there is *some* $x \in L$ with $|x| \geq n$ such that for *all* u, v and w where $uvw = x$, $|uv| \leq n$ and $|v| \geq 1$ there is *some* i for which $uv^iw \notin L$.

A useful way to think of this is as a game between you, who are trying to prove that the pumping lemma fails for L, and an adversary that is trying to prove that it holds. Expressed formally, the lemma says ('$\forall$' and '$\exists$' should be read "for all" and "there exists", respectively):

$$
\begin{array}{l}
(\forall L)[L \text{ regular } \Rightarrow \\
\quad (\exists n)[\\
\quad\quad (\forall x)[x \in L \text{ and } |x| \geq n \Rightarrow \\
\quad\quad\quad (\exists u, v, w)[x = uvw \text{ and} \\
\quad\quad\quad\quad |uv| \leq n \text{ and} \\
\quad\quad\quad\quad |v| \geq 1 \text{ and} \\
\quad\quad\quad\quad (\forall i \geq 0)[uv^iw \in L]]]]].
\end{array}
$$

Every '$\forall$' is your choice—the lemma should be true for any value you choose. Every '$\exists$' is your opponent's choice—they need only exhibit some value for which the lemma is true. Each move can depend on all the choices made prior to it. The game starts with your choice of the L you wish to prove to be non-regular. You opponent then chooses some n, you choose a string $x \in L$ of length at least n, etc. You win if, at the end of this process, you can choose i such that $uv^iw \notin L$. To establish that the lemma is not satisfied by L you have to show that no matter which choices your adversary makes, you can always have a winning choice of x and i, that is, you must give a strategy, accounting for all possible choices of your opponent, that always leads to a win for you.

Of course, your strategy at each step will depend on the choices your opponent has made.

What we end up with is a proof by contradiction. For instance:

To show that $L_{ab} = \{a^j b^j \mid j \geq 0\}$ is not regular.

Proof: Suppose, by way of contradiction, L_{ab} is regular. (Your choice of L.)
Let n be the constant of the pumping lemma. (Your adversary chooses n. You can make no assumptions about n but you will base your subsequent choices on its value.)
Let $x = a^n b^n$. (Your choice of x. Note that it depends on n. In fact, if your choice does not depend n, if n is not a parameter of your definition of x, then the proof will almost certainly fail.)
Since $x \in L_{ab}$ and $|x| \geq n$, by the pumping lemma there must be some u, v and w such that $x = uvw$, $|uv| \leq n$, $|v| \geq 1$ and $uv^iw \in L_{ab}$ for all i. (You opponent chooses how to split x into uvw subject only to the conditions that $|uv| \leq n$ and $|v| \geq 1$. Your strategy must account for all ways of doing this—all you can assume about u, v and w is that they meet the conditions of the lemma.)
Since $|uv| \leq n$, both u and v must fall within the first n symbols of x. Thus, u and v must consist only of 'a's. Furthermore, v must include at least one 'a', since $|v| \geq 1$. Thus, there must be some j, k and l such that

$$u = a^j, \qquad v = a^k, \qquad w = a^l b^n, \qquad j + k + l = n, \text{ and } k \geq 1.$$

But suppose $i = 0$. (Your choice of i.) Then

$$uv^0w = uw = a^j a^l b^n$$

where $j + l = n - k$ which is strictly less than n. Thus $uv^iw \notin L_{ab}$ for $i = 0$ and the pumping lemma does not hold for L_{ab}, contradicting our assumption that L_{ab} was regular. ⊣

In this case our choice of x restricted the adversary's choice of uvw enough that all choices can be treated with a single argument. In general this will not be the case and, even if we can use the same i in all choices, we will have to account separately for each alternative way of dividing x.

Example: Let $L_4 \subseteq \{a,b\}^*\{c,d\}^*$ be the language in which a string w is in L_4 iff the number of occurrences of the substring *ab* in w is greater than or equal to the number of occurrences of the substring *cd* in w. Prove that L_4 is non-regular using the pumping lemma.

Proof: Suppose, for contradiction, that L_4 is regular. Let n be the constant of the pumping lemma. Let $x = (ab)^n(cd)^n$. Let $|w|_{ab}$ denote the number of occurrences of the substring *ab* in w. Similarly for $|w|_{cd}$. Then $x \in L_4$, $|x| > n$, and $|x|_{ab} = |x|_{cd} = n$.

By the pumping lemma, $x = uvw$, where $|uv| \leq n$, $|v| \geq 1$ and $uw \in L_4$. Consequently, uv is a prefix of $(ab)^n$.
We must now account for every way in which a prefix of $(ab)^n$ can be divided into uv with $|v| \geq 1$. Any way of dividing these possibilities into cases is acceptable, *as long as it is exhaustive.* One way that works well for this language is to note that v must start with either an 'a' or a 'b' and, similarly, must end with one or the other. Thus, there are four cases:

$$v = a(ba)^j \qquad v = a(ba)^j b \qquad v = b(ab)^j a \text{ or } v = b(ab)^j, \qquad j \geq 0.$$

By cases, then:

$$\begin{array}{llllll}
u = (ab)^i & v = a(ba)^j & w = b(ab)^k(cd)^n & \Rightarrow & |uw|_{ab} & = & n - 1 - |v|_{ab} \\
u = (ab)^i & v = a(ba)^j b & w = (ab)^k(cd)^n & \Rightarrow & |uw|_{ab} & = & n - |v|_{ab} \text{ and } |v|_{ab} > 0. \\
u = (ab)^i a & v = b(ab)^j a & w = b(ab)^k(cd)^n & \Rightarrow & |uw|_{ab} & = & n - 1 - |v|_{ab} \\
u = (ab)^i a & v = b(ab)^j & w = (ab)^k(cd)^n & \Rightarrow & |uw|_{ab} & = & n - 1 - |v|_{ab}
\end{array}$$

Thus in all cases $|uw|_{ab} < |uw|_{cd} = n$, $uw \notin L_4$ and L_4 is not regular. $\dashv$

36. Prove that $L_5 \overset{\text{def}}{=} \{w \in \{a,b\}^* \mid |w|_a < |w|_b\}$ is not regular (where $|w|_a$ is the number of 'a's occurring in w).

37. Prove that the following language L_6 is not regular.

$$L_6 \overset{\text{def}}{=} \{w \in \{a,b\}^* \mid |w|_a < |w|_b \text{ if } |w|_a \text{ even, } |w|_a > |w|_b \text{ otherwise } \}$$

11 The Myhill-Nerode Theorem

As we have defined them, in every DFA there is a path from the start state to some state of the DFA for every string in Σ^*. Since there are infinitely many strings but only finitely many states many strings must lead to the same state. Recall that the state represents all the information that the DFA has about the string leading to that state, so, in a strong sense it is the nature of DFAs that they are *forgetful*—they are unable to remember a complete description of the strings they scan and must, inevitably, fail to distinguish some strings. The content of the Myhill-Nerode Theorem, the topic of this section, is that this partitioning of Σ^* into finitely many classes of indistinguishable strings is characteristic of the regular languages.

Suppose L is regular. Then there is some DFA $\mathcal{A}$ that accepts it, $L = L(\mathcal{A})$. The approach we have taken to proving that a DFA accepts a language is to identify each state with an invariant, a property that is characteristic of the strings that label paths from the start state to that state. These invariants define a relation on strings: two strings are related iff they satisfy the same invariant, which is to say iff they lead to the same state. We will refer to this relation as $R_{\mathcal{A}}$:

$$R_{\mathcal{A}} \stackrel{\text{def}}{=} \{\langle w, v\rangle \mid \hat{\delta}(q_0, w) = \hat{\delta}(q_0, v)\}$$

and will use $R_{\mathcal{A}}$ as an 'infix' relation symbol:

$$wR_{\mathcal{A}}v \Leftrightarrow \langle w, v\rangle \in R_{\mathcal{A}}.$$

Two strings are equivalent, from the point of view of $\mathcal{A}$, iff they are related by $R_{\mathcal{A}}$. We can verify that $R_{\mathcal{A}}$ is an *equivalence relation*—it is *reflexive*: $wR_{\mathcal{A}}w$ for all w; it is *symmetric*: if $wR_{\mathcal{A}}v$ then $vR_{\mathcal{A}}w$ as well; and it is *transitive*: $uR_{\mathcal{A}}v$ and $vR_{\mathcal{A}}w$ implies $uR_{\mathcal{A}}w$; so it is consistent with this interpretation.

Let

$$[w]_{R_{\mathcal{A}}} \stackrel{\text{def}}{=} \{v \mid wR_{\mathcal{A}}v\}.$$

This is the *equivalence class of w wrt $R_{\mathcal{A}}$*, the set of all strings equivalent to w in the $R_{\mathcal{A}}$ sense. Note that, as with all equivalence classes, every string $w \in \Sigma^*$ is in some class (namely $[w]_{R_{\mathcal{A}}}$) and no string is in more than one ($w \in [u]_{R_{\mathcal{A}}}$ and $w \in [v]_{R_{\mathcal{A}}}$ implies $[u]_{R_{\mathcal{A}}} = [v]_{R_{\mathcal{A}}}$). Thus the equivalence classes of $R_{\mathcal{A}}$ *partition* Σ^*—they are disjoint and their union is Σ^*.

Now, we potentially have many names for each equivalence class since if $wR_{\mathcal{A}}v$ then $[w]_{R_{\mathcal{A}}} = [v]_{R_{\mathcal{A}}}$. We can capture the set of all classes by choosing an (arbitrary) canonical representative for each class. This set of representatives is referred to as an *index set* or *spanning set* (we will denote it with I) and

$$\Sigma^* = \bigcup_{w \in I} [[w]_{R_{\mathcal{A}}}] .$$

Given that $R_{\mathcal{A}}$ is defined by the automaton $\mathcal{A}$, how big is its index set? Certainly, there can be no more equivalence classes than there are states of $\mathcal{A}$ (there can be fewer, since some states of $\mathcal{A}$ may be inaccessible from q_0). Thus the number of equivalence classes wrt $R_{\mathcal{A}}$ is finite; we say that $R_{\mathcal{A}}$ has *finite index* (or is spanned by a finite set).

Since the finiteness of the number of equivalence classes of $R_{\mathcal{A}}$ is a consequence of the key characteristic of DFAs—the finiteness of their state sets—it should seem plausible that the existence of *some* relation $R_{\mathcal{A}}$ with finite index is a key property of regular sets. To pin this down we need to look a little more closely at the nature of $R_{\mathcal{A}}$ and its relationship to L.

First, note that, since two strings w and u are related by $R_{\mathcal{A}}$ iff the paths (from the start state) they label lead to the same state, it must be the case that if we extend them both with the same string we will obtain, again, equivalent strings, i.e.,

$$wR_{\mathcal{A}}u \Rightarrow (\forall v)[wvR_{\mathcal{A}}uv].$$

We say that $R_{\mathcal{A}}$ is *right invariant* wrt concatenation.

Finally, we note that

$$L = \bigcup_{\hat{\delta}(q_0,w) \in F} [[w]_{R_{\mathcal{A}}}] ,$$

which is to say, L is the union of some of the equivalence classes of Σ^* wrt $R_{\mathcal{A}}$.

When we put these all together, we get a lemma.

Lemma 15 *If a language $L \subseteq \Sigma^*$ is regular then it is the union of some of the equivalence classes of a right invariant equivalence relation on Σ^* which is of finite index.*

Note that, while given $\mathcal{A}$ accepting L we can immediately produce $R_{\mathcal{A}}$, the lemma is actually independent of $\mathcal{A}$. If we know that L is regular then we know that some relation with the properties we have established for $R_{\mathcal{A}}$ exists, since some $\mathcal{A}$ does. Intuitively, the converse ought to be true as well—if such an equivalence relation exists then we ought to be able to construct a DFA using the equivalence classes of the relation as states; the existence of such a relation ought to imply that L is regular. And, in fact, it does but it is useful to prove this in a somewhat round-about way.

The fact that $R_{\mathcal{A}}$ is right invariant wrt concatenation implies that

$$wR_{\mathcal{A}}u \Rightarrow (\forall v)[wv \in L \Leftrightarrow uv \in L].$$

Let's consider the relation

$$R_L \stackrel{\text{def}}{=} \{\langle w, u\rangle \mid (\forall v)[wv \in L \Leftrightarrow uv \in L]\}.$$

Note that, although this is also an equivalence relation, it is *not* the same relation as $R_{\mathcal{A}}$ because it may be the case that $\mathcal{A}$ distinguishes two strings w and u even though wR_Lu—w and u may lead to different states of $\mathcal{A}$, even though $\mathcal{A}$ behaves the same (in the sense of accepting or not) on all strings extending them. However, as we just saw, if $wR_{\mathcal{A}}u$ then wR_Lu, thus every equivalence class wrt $R_{\mathcal{A}}$ is a subset of an equivalence class wrt R_L; the equivalence classes of $R_{\mathcal{A}}$ do not *break* the equivalence classes of R_L, rather they partition them. We say that $R_{\mathcal{A}}$ is a *refinement* of R_L—every equivalence class wrt R_L is the union of the equivalence classes of $R_{\mathcal{A}}$ it intersects. Thus, the number of equivalence classes of R_L can be no greater than the number of equivalence classes of $R_{\mathcal{A}}$. Consequently, the number of equivalence classes of Σ^* wrt R_L is also finite. R_L has finite index. This leads to our second lemma.

Lemma 16 *Suppose $L \subseteq \Sigma^*$. Let $R_L \stackrel{\text{def}}{=} \{\langle w, u\rangle \mid (\forall v)[wv \in L \Leftrightarrow uv \in L]\}$. If L is regular then R_L has finite index.*

This follows from the previous lemma and the fact that right invariance of $R_{\mathcal{A}}$ implies that $R_{\mathcal{A}}$ refines R_L.

We are close, now, to having characterizations of the regular languages in terms of both $R_{\mathcal{A}}$ and R_L. If we can show that whenever R_L has finite index then L is regular we will get not only that this characterizes the regular sets but also that being the union of some of the classes of a right invariant equivalence relation of finite index does, since it is implied by regularity and implies finiteness of index of R_L.

To obtain this result, we will follow the idea we sketched for $R_{\mathcal{A}}$—we will construct a DFA accepting L using the equivalence classes of Σ^* wrt R_L as our state set.

Let $Q' = \{[w]_{R_L} \mid w \in \Sigma^*\}$. (Note that while each of these states is a potentially infinite set, the number of states is finite. To avoid even this vestigial infiniteness, we could take Q' to be any index set for R_L, but this definition is slightly simpler.)

Let $\delta'([w]_{R_L}, \sigma) = [w\sigma]_{R_L}$.

We need to verify that this is, in fact, well defined. In particular, we need to show that for all u and σ, if $u \in [w]_{R_L}$ then $u\sigma \in [w\sigma]_{R_L}$. (In other words, we need to show that R_L is right invariant, but this does not follow from right invariance of $R_\mathcal{A}$ since uR_Lw does not imply $uR_\mathcal{A}w$.) To see that this is the case, suppose uR_Lw. Then, for all v, $uv \in L \Leftrightarrow wv \in L$. And, in particular (letting $v = \sigma v'$), for all v', $u\sigma v' \in L \Leftrightarrow w\sigma v' \in L$. Thus, $u\sigma R_L w\sigma$.

Finally, let $q'_0 = [\varepsilon]_{R_L}$ and $F' = \{[w]_{R_L} \mid w \in L\}$.

Lemma 17 *Let $\mathcal{A}' = \langle Q', \Sigma, q'_0, \delta', F'\rangle$, where Q', q'_0, δ' and F' are as described above. Then $L = L(\mathcal{A}')$.*

38. Prove that, for all $w \in \Sigma^*$, $\hat{\delta}'(q'_0, w) = [w]_{R_L}$.

39. Use this to prove the lemma.

Putting the three lemmas together we get:

Theorem 2 (Myhill-Nerode) *The following are equivalent:*

- *L is regular.*
- *L is the union of some of the equivalence classes of a right invariant equivalence relation of finite index.*
- *The relation $R_L \overset{\text{def}}{=} \{\langle w, u\rangle \mid (\forall v)[wv \in L \Leftrightarrow uv \in L]\}$ has finite index.*

Note that with R_L we have abandoned the notion of DFA entirely. This is a relation that is defined directly in terms of the strings that are and are not in the language, independent of any particular mechanism for defining or accepting it.

11.1 Using Myhill-Nerode to Prove Non-Regularity

The fact that a language L is regular iff R_L has finite index gives us another approach to proving languages are not regular. If we can show that there must be infinitely many equivalence classes of Σ^* wrt R_L then L cannot be regular. This is often simpler than using the Pumping Lemma.

The most direct approach to doing this is to give an infinite sequence of strings in Σ^* all of which are distinct wrt to R_L. Since they each must be in a distinct equivalence class and since there are infinitely many of them, R_L cannot have finite index.

Example: To show that

$$L_{ab} = \{a^i b^i \mid i \geq 0\}.$$

is not regular, consider the sequence of strings

$$\langle a^0, a^2, \ldots, a^i, \ldots \mid i \geq 0\rangle.$$

We claim that every string in this sequence is distinct wrt R_L from every other string in the sequence. To see this, consider a^i and a^j for $i \neq j$. Then $a^i b^i \in L_{ab}$ while $a^j b^i \notin L_{ab}$. Thus, b^i witnesses that a^i and a^j are *not* related by R_L. Since i and j are arbitrary, every string in the sequence is distinct, wrt R_L, from every other string in the sequence; no two share the same equivalence class. It follows that there must be at least as many equivalence classes of R_L as there are strings in the sequence; R_L cannot have finite index.

40. Consider, again, the problem of specifying schedules for a machine tool (see Section 7.3). Suppose, in this instance, that there is just a single type of part which requires two operations A_1 and A_2 to complete. These can be done in any order, although we will assume that operations always complete partially completed parts if they can. Thus, at any given time the partially completed parts will all be waiting for the same operation (although which operation can change over time). Assume, further, that there is an unbounded amount of space to store partially completed parts; the only constraint on a schedule is that every part gets completed.

    (a) Describe this language.

    (b) Use Myhill-Nerode to show that the set of feasible schedules, given these constraints, is not regular.

11.2 Using Myhill-Nerode to Prove Regularity

The Myhill-Nerode theorem gives us a characterization of the regular languages—a language is regular *iff* the second and third part of theorem are satisfied.

Thus, in contrast to the Pumping Lemma, we can use Myhill-Nerode not only to prove languages are not regular, but also to prove that languages *are* regular. In fact, one of the attractive features of this approach is the fact that if one's attempt to prove a language *is not* regular fails then one is likely to be well along the way to proving that the language *is* regular.

While it is possible to use either $R_{\mathcal{A}}$ or R_L to prove a language is regular, using R_L is usually much simpler. The idea is to consider the way in which Σ^* is partitioned by R_L and argue that there are only finitely many partitions.

Example: Consider, again, the example of Section 7.3. Let L_2 be the set of feasible schedules under the constraints given there. Following the Myhill-Nerode Theorem, let

$$wR_{L_2}u \overset{\text{def}}{\Longleftrightarrow} (\forall v)[wv \in L_2 \Leftrightarrow uv \in L_2].$$

Now, for any strings $w, v \in \{A_1, A_2, B_1, B_2\}^*$, it will be the case that $wv \in L_2$ iff

- *v completes w*:

$$|w|_{A_1} - |w|_{A_2} = |v|_{A_2} - |v|_{A_1} \text{ and } |w|_{B_1} - |w|_{B_2} = |v|_{B_2} - |v|_{B_1}.$$

- *wv* does not satisfy:

$$\text{FAIL}(x) \overset{\text{def}}{\Longleftrightarrow} x = x'x'' \text{ where } \left||x'|_{A_1} - |x'|_{A_2}\right| + \left||x'|_{B_1} - |x'|_{B_2}\right| > 2,$$

The first condition says that every part pending at the end of the schedule w will be completed during the schedule v. The second says that there will never be more than two pending parts during this process.

Consider, now, an arbitrary strings in $w, u \in \{A_1, A_2, B_1, B_2\}^*$. Under what circumstances will it be the case that $wR_{L_2}u$? Well, if FAIL(w) is true, then there will be no v for which $wv \in L_2$. On the other hand, if FAIL(u) is not true, then there is at least one v for which $uv \in L_2$—the one that simply completes the parts outstanding at the end of u. Thus, if FAIL(w) and FAIL(u) then $wR_{L_2}u$, but if FAIL(w) and not FAIL(u) or *vice versa* then *not* $wR_{L_2}u$. It follows that the class of all strings that satisfy FAIL is one of the equivalence classes of $\{A_1, A_2, B_1, B_2\}^*$ wrt R_{L_2}. (This should not be a surprise, the class corresponds to the state Fail of the DFA we constructed for this language.)

It remains to consider what determines if two strings neither of which satisfy FAIL are related by R_{L_2}. First of all, it should be clear that a schedule v will complete both w and u iff the set of parts outstanding at the end of w is the same as the set outstanding at the end of u. For example,

$$(|w|_{A_1}-|w|_{A_2} = |v|_{A_2}-|v|_{A_1} \text{ and } |u|_{A_1}-|u|_{A_2} = |v|_{A_2}-|v|_{A_1}) \Leftrightarrow |w|_{A_1}-|w|_{A_2} = |u|_{A_2}-|u|_{A_1}.$$

Moreover, if the set of parts outstanding at the end of w is the same as the set outstanding at the end of u then, for all v, wv will satisfy FAIL iff uv satisfies FAIL. Conversely, it is not hard to see that if the set of parts outstanding at the end of w and u are *not* the same then there will be a schedule v which completes one while FAILing on the other.

Thus, for $w, u \in \{A_1, A_2, B_1, B_2\}^*$, $wR_{L_2}u$ iff

- FAIL(w) and FAIL(u), or
- Not FAIL(w) and not FAIL(u) and $|w|_{A_1} - |w|_{A_2} = |u|_{A_2} - |u|_{A_1}$ and $|w|_{B_1} - |w|_{B_2} = |u|_{B_2} - |u|_{B_1}$.

It remains, to count how many equivalence classes this relation generates. There will be one for strings that satisfy fail plus one for each pair of values of $|w|_{A_1} - |w|_{A_2}$ and $|w|_{B_1} - |w|_{B_2}$ for w that do not satisfy FAIL. Clearly, since w does not satisfy FAIL,

$$-2 \leq |w|_{A_1} - |w|_{A_2}, |w|_{B_1} - |w|_{B_2} \leq 2.$$

It follows that there can be no more than $5 \cdot 5 = 25$ such pairs and $25 + 1 = 26$ classes altogether. (There are actually just 14.) Thus, R_{L_2} has finite index and, by the Myhill-Nerode Theorem, L_2 is regular.

41. Using the Myhill-Nerode Theorem, prove the language L_3 of Problem 25 is regular.

11.3 Minimization of DFAs

Recall that we showed above that $R_{\mathcal{A}}$ was a refinement of R_L; if two strings are related by $R_{\mathcal{A}}$ they are necessarily related by R_L. The converse of this is not true—it may be the case that $\mathcal{A}$ distinguishes two strings, that the paths from the initial state labeled with the strings lead to distinct states in $\mathcal{A}$, even though any string labeling a path to some final state from one of those states also labels a path to some final state from the other and *vice versa.* But, certainly, there is no need to distinguish these states. Since the behavior of the automaton is the same from both states, it should be possible to merge them into a single state. (This is not as immediate as it seems. The fact that a path labeled with some string leads to a final state from the one if and only if it leads to some final state from the other does not imply that they lead to the *same* final state. We may, in general, have to merge a number of states in order to preserve a deterministic transition function.)

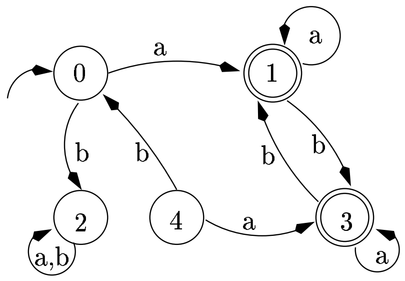
Thus, if there is more than a single equivalence class wrt $R_{\mathcal{A}}$ partitioning any equivalence class wrt R_L the automaton $\mathcal{A}$ includes more states than necessary; there is a simpler automaton that accepts the same language. Consider, now, whether any of the classes wrt R_L are redundant. These are, after all, the classes we chose as the state set of the automaton we constructed in the proof of the Myhill-Nerode Theorem. (For that DFA, in fact, $R_{\mathcal{A}} = R_L$.) Is it possible to merge any of these states, to construct a simpler automaton that accepts the same language?

Suppose $[w]_{R_L} \neq [u]_{R_L}$. Then, by the definition of R_L there must be some string v such that $wv \in L$ while $uv \notin L$ or *vice versa.* But if we were to merge these two states the automaton would have to either accept both wv and uv or reject them both. Thus, if $[w]_{R_L} \neq [u]_{R_L}$ then every DFA that accepts L will need to distinguish them. It follows that the DFA we constructed on the equivalence classes wrt R_L is, in fact, *minimal*—there is no DFA with fewer states that accepts L. Moreover, while there will be other DFAs with the same number of states that accept L (since we could take any finite set of the same size to be Q), every one of these will have to distinguish exactly the same sets of strings; necessarily $R_{\mathcal{A}} = R_L$ for all of them. It follows, then, that the only distinction between the minimal DFAs accepting L is the labeling of the states. We say that they are *isomorphic.*

Lemma 18 *The DFA constructed on R_L is minimal in the size of its state set among DFAs accepting L. Moreover, up to isomorphism, it is the* unique *minimal DFA accepting L.*

This gives us a technique for minimizing DFAs, for eliminating redundant states. As DFAs employed in applications can get quite large, such minimalization can have a significant effect on efficiency. The idea is to identify classes of $R_{\mathcal{A}}$, states of $\mathcal{A}$, that are *indistinguishable* wrt R_L, where a pair of states $q, p \in Q$ are *distinguished* wrt R_L iff there is a string v which leads to a final state from one and to a non-final state from the other, i.e., iff $\delta(q,v) \in F$ and $\delta(p,v) \notin F$, or *vice versa.* As we noted above, whether such a string exists is not necessarily obvious. The length of the string v could, potentially, be quite long. The approach we will take to identifying pairs of states that are distinguished wrt R_L is to iterate through $i \geq 0$ identifying all those pairs that are distinguished by strings of length i and then using those to identify those pairs distinguished by strings of length $i+1$, etc. This leads to a dynamic programming algorithm, which we will lay out by example.

Let $\mathcal{A}$ be the DFA:



We will construct a table relating pairs of states in which the entry in the q row and the p column will be non-empty iff we have distinguished states q and p. Since the relation of being distinguished is symmetric, we need only the lower triangular of this table.

1				
2				
3				
4				
	0	1	2	3

In the first iteration $i = 0$. We will mark each pair of states that are distinguished by a string of length 0, which is to say we will mark the entry for $\langle p,q \rangle$ iff $\hat{\delta}(p,\varepsilon) \in F$ and $\hat{\delta}(q,\varepsilon) \notin F$ or *vice versa.* In other words, we distinguish every state in F from every state not in F.

1	ε			
2		ε		
3	ε		ε	
4		ε		ε
	0	1	2	3

(We have marked them with the string that distinguishes them.)

In the subsequent iterations we identify states that are distinguished by increasingly long strings. A pair of states $\langle p, q\rangle$ will be distinguished by a string of length $i+1$ iff there is some $\sigma \in \Sigma$ for which the state reached from p on σ and the state reached from q on σ are distinguished by a path of length i, i.e., if $\delta(p,\sigma)$ and $\delta(q,\sigma)$ were distinguished in iteration i. Thus, in each iteration, we work our way through the table marking each entry $\langle p, q\rangle$ for which the entry $\langle \delta(p,\sigma), \delta(q,\sigma)\rangle$ (or *vice versa*) is already marked.

1	ε			
2	a	ε		
3	ε		ε	
4		ε	a	ε
	0	1	2	3

$\delta(0,a) = 1$ $\delta(2,a) = 2$
$\delta(2,a) = 2$ $\delta(4,a) = 3$

1	ε			
2	a	ε		
3	ε		ε	
4	ba	ε	a	ε
	0	1	2	3

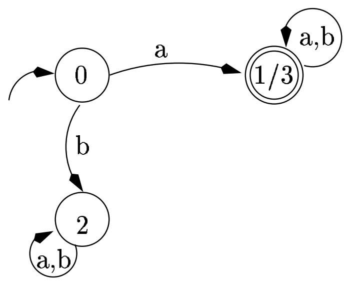
$\delta(0,b) = 2$ $\delta(4,b) = 0$

We repeat this until some iteration fails to distinguish any new pairs of states. It should be clear that from that point on no more pairs will be distinguished. That all distinguishable pairs will have been marked at that point follows from the invariant:

> An entry $\langle p, q\rangle$ will be marked in the table at iteration i iff there is a string v of length no greater than i for which $\hat{\delta}(p, v) \in F$ while $\hat{\delta}(q, v) \notin F$ or *vice versa*

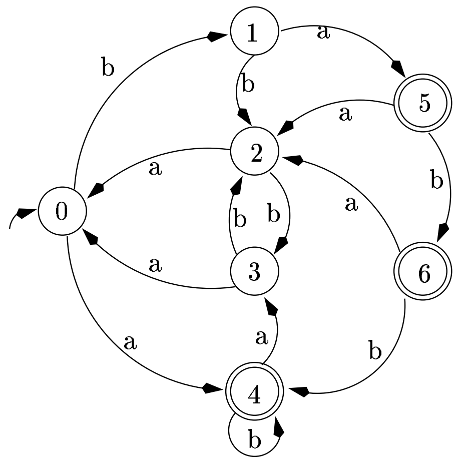
which can be easily established by induction on i. That the algorithm always terminates follows from the fact that the table is finite—one cannot distinguish any more pairs than there are entries in the table, thus, the algorithm converges after no more than that many iterations.

In the example, the pair $\langle 3, 1\rangle$ is left unmarked, thus, these states are indistinguishable wrt R_L and can be merged. Note, also, that, while state 4 is distinguished from every other state it is unreachable from the initial state and is, therefore, useless. We can simply eliminate it.



Note that, in merging state p and q the new transition function will be well defined iff $\delta(p, \sigma)$ and $\delta(q, \sigma)$ are equivalent states. The algorithm guarantees this will be the case (why?).

42. Minimize the following DFA.



12 Closure Properties of the Class of Regular Languages

The pumping lemma gives a kind of closure property for individual regular languages: if the language includes a string of a particular form then it includes all strings of a related form. In this section we will look at some of the closure properties of the *class* of regular languages—properties of the form: if $L_1, \ldots, L_k$ are in that class (are regular) then the languages formed from the L_i by some particular operation are all in the class as well.

12.1 Boolean Operations

Theorem 3 *The class of regular languages is closed under union, concatenation, and Kleene closure.*

Proof: This follows immediately from the definition of regular languages. $\dashv$
Note that we actually gave constructive proofs of these closure results as part of the proof that every regular language is accepted by some DFA.

Theorem 4 *The class of regular languages is closed under intersection.*

Proof (sketch)**:** Suppose L_1 and L_2 are regular. Then there are DFAs $\mathcal{M}_1 = \langle Q, \Sigma, \delta_1, q_0, F_1\rangle$ and $\mathcal{M}_2 = \langle P, \Sigma, \delta_2, p_0, F_2\rangle$ such that $L_1 = L(\mathcal{M}_1)$ and $L_2 = L(\mathcal{M}_2)$. We construct $\mathcal{M}'$ such that $L(\mathcal{M}') = L_1 \cap L_2$. The idea is to have $\mathcal{M}'$ run $\mathcal{M}_1$ and $\mathcal{M}_2$ in parallel—keeping track of the state of both machines. It will accept a string, then, iff both machines reach a final state on that string.

Let $\mathcal{M}' = \langle Q \times P, \Sigma, \delta', \langle q_0, p_0\rangle, F_1 \times F_2\rangle$, where

$$\delta'(\langle q, p\rangle, \sigma) = \langle \delta_1(q,\sigma), \delta_2(p,\sigma)\rangle .$$

Then $\hat{\delta}'(\langle q,p\rangle, w) = \left\langle \hat{\delta}_1(q,w), \hat{\delta}_2(p,w)\right\rangle$. (You should prove this; it is an easy induction on the structure of w.) It follows then that

$$\begin{aligned}
w \in L(\mathcal{M}') &\Leftrightarrow \hat{\delta}'(\langle q_0, p_0\rangle, w) \in F_1 \times F_2 \\
&\Leftrightarrow \hat{\delta}_1(q_0, w) \in F_1 \text{ and } \hat{\delta}_2(p_0, w) \in F_2 \\
&\Leftrightarrow w \in L_1 \text{ and } w \in L_2 \\
&\Leftrightarrow w \in L_1 \cap L_2.
\end{aligned}$$

$\dashv$

Corollary 1 *The class or regular languages is closed under relative complement.*

Since $w \in L_1 \setminus L_2$ iff $w \in L_1$ and $w \notin L_2$ iff $\hat{\delta}_1(q_0, w) \in F_1$ and $\hat{\delta}_2(p_0, w) \notin F_2$ iff $\hat{\delta}'(\langle q_0, p_0 \rangle, w) \in F_1 \times (P \setminus F_2)$, we can use essentially the same construction changing only F' to $F_1 \times (P \setminus F_2)$.

Theorem 5 *The class of regular languages is closed under complement.*

Proof (sketch)**:** Following the insight of the corollary, $w \in \overline{L_1}$ iff $w \notin L_1$ iff $\hat{\delta}_1(q_0, w) \notin F_1$. Thus we can let $\mathcal{M}' = \langle Q, \Sigma, \delta_1, q_0, Q \setminus F_1 \rangle$, that is $\mathcal{M}$ with the set of final states complemented wrt Q. ⊣

Note that if we had proved closure under complement first we could have gotten closure under intersection using DeMorgan's Theorem.

$$L_1, L_2 \text{ reg.} \Rightarrow \overline{L_1}, \overline{L_2} \text{ reg.} \Rightarrow \overline{L_1} \cup \overline{L_2} \text{ reg.} \Rightarrow \overline{\overline{L_1} \cup \overline{L_2}} \text{ reg.} \Rightarrow L_1 \cap L_2 \text{ reg.}.$$

Definition 45 *A class of languages is closed under* Boolean Operations *iff it is closed under union, intersection, and relative complement.*

Corollary 2 *Any class of languages closed under relative complement and either union or intersection is closed under Boolean operations.*

Corollary 3 *The class of regular languages is closed under Boolean operations.*

12.2 Using Closure Properties to Prove Regularity

The fact that regular languages are closed under Boolean operations simplifies the process of establishing regularity of languages; in essence we can augment the regular operations with intersection and complement (as well as any other operations we can show preserve regularity). All one need do to prove a language is regular, then, is to show how to construct it from "obviously" regular languages using any of these operations. (A little care is needed about what constitutes "obvious". The safest thing to do is to take the language back all the way to $\emptyset$, $\{\varepsilon\}$, and the singleton languages of unit strings.)

Example: Let $L \subseteq \{a, b\}^*$ such that

- 'aa' never occurs in any string in L,
- if 'ab' occurs anywhere in a string in L then 'ba' also occurs somewhere in that string.

To show that L is regular, note first that L is the intersection of two languages: one in which only the first property (no 'aa') is enforced and one in which only the second ('ab' implies 'ba') is.

$$L = L_1 \cap L_2,$$

where L_1 is the set of strings over $\{a, b\}$ in which 'aa' never occurs and L_2 is the set in which 'ba' occurs whenever 'ab' does.

$$L_1 = \overline{L_3},$$

where L_3 is the set of strings over $\{a, b\}$ in which 'aa' does occur.

$$L_3 = L((a + b)^* aa(a + b)^*).$$

L_2 is the set of strings over $\{a, b\}$ in which either 'ab' does not occur or 'ba' does ($P \Rightarrow Q \equiv \neg P \vee Q$).

$$L_2 = L_4 \cup L_5,$$

where L_4 is the set of strings over $\{a, b\}$ in which 'ab' never occurs and L_5 is the set in which 'ba' does.

$$L_4 = \overline{L_6}, \qquad L_6 = L((a + b)^* ab(a + b)^*).$$

and

$$L_5 = L((a + b)^* ba(a + b)^*).$$

Thus

$$L = \overline{L_3} \cap (\overline{L_6} \cup L_5)$$

and each of L_3, L_6 and L_5 are regular. Hence L is regular as well.

43. Using this approach, show that the set of strings over $\{a, b\}$ in which the number of 'a's is divisible by three but not divisible by two is regular.

44. Starting with simple automata for your "obviously" regular languages and using the constructions of the proofs of the closure properties, build a DFA for this language.

45. Consider the two languages:

    L_a: The set of strings over $\{a, b\}$ in which the last symbol is not 'b'.

    L_b: The set of strings over $\{a, b\}$ in which the last symbol is not 'a'.

    Using the approach of the previous problem, construct a DFA accepting the language of strings that satisfy both of these descriptions.

46. What is that language? Explain why it is not empty.

12.3 Quotient and Prefix

Definition 46 *The (right)* quotient *of a language L_1 wrt L_2 (both over Σ) is*

$$L_1/L_2 \overset{\text{def}}{=} \{w \in \Sigma^* \mid (\exists v \in L_2)[wv \in L_1]\}.$$

So the quotient of L_1 wrt L_2 is the set of prefixes one is left with when one removes suffixes of strings in L_1 that are found in L_2.

Example: Let $L_1 = L(a^*(bc)^*)$ and $L_2 = L((cb^*)^+)$. Then

$$L_1/L_2 = \{w \in \{a, b, c\}^* \mid (\exists v \in L((cb^*)^+))[wv \in L(a^*(bc)^*)]\}.$$

If $wv \in L_1$ then $wv = a^i(bc)^j$ for some $i, j \in \mathbb{N}$.
If $v \in L_2$ then $v = (cb^{k_1})(cb^{k_2}) \cdots (cb^{k_l})$ for some $k_1, k_2, \ldots, k_l \in \mathbb{N},\ l > 0$.
It follows that

$$wv = a^i b(cb)^m c = a^i b(cb)^{m-l-1}(cb^1)^{l-1}(cb^0) \text{ and } w = a^i b(cb)^{m-l-1},\ m-l-1 \geq 0.$$

Thus

$$L_1/L_2 = L(a^*b(cb)^*).$$

47. Let $L_1 = L(a^*ba^*)$ and $L_2 = L(b^*a)$. What is L_1/L_2?

48. Let $L_3 = L(ba^*b)$ and L_1 remain the same. What is L_1/L_3?

Theorem 6 *The class of regular languages is closed under quotient with* arbitrary languages.

That is, as long as L_1 is regular, L_2 can be any language whatsoever and L_1/L_2 will be regular. (It is not required that it even be possible to effectively decide if a given word is in L_2.)

Proof (sketch)**:** Let $L_1 = L(\mathcal{M}_1)$ and $\mathcal{M}_1 = \langle Q, \Sigma, \delta, q_0, F\rangle$.
Let $\mathcal{M}' = \langle Q, \Sigma, \delta, q_0, F'\rangle$ where

$$F' = \{q \in Q \mid (\exists v \in L_2)[\hat{\delta}(q, v) \in F]\}.$$

It is easy to show, then, that $w \in L(\mathcal{M}') \Leftrightarrow w \in L_1/L_2$. $\dashv$

49. Show it.

Note that if it is not possible to effectively decide if $v \in L_2$ then we will not be able to effectively decide if $q \in F'$. But there are only $2^{\mathbf{card}(Q)}$ subsets of Q. One of these is the right one. Thus there is *some* DFA that recognizes L_1/L_2, although we may not be able to effectively decide which one. Of course, if L_2 is regular we can tell if $w \in L_2$ and the construction is effective.

Corollary 4 *The class of regular languages is closed under the operation*

$$\text{Prefix}(L) \overset{\text{def}}{=} \{w \in \Sigma^* \mid (\exists v \in \Sigma^*)[wv \in L]\}.$$

Since $\text{Prefix}(L) = L/\Sigma^*$.

50. Suppose L is any nonempty language. What is Σ^*/L?

51. What is $L/\emptyset$?

52. What is $L/\{\varepsilon\}$?

53. Suppose L_1, L_2 and L_3 are arbitrary languages. Show that

$$L_1/(L_2 \cup L_3) = (L_1/L_2) \cup (L_1/L_3)$$

and that

$$L_1/(L_2 \cap L_3) = (L_1/L_2) \cap (L_1/L_3).$$

54. Suppose, again, that L_1, L_2 and L_3 are arbitrary languages. What is $L_1/(L_2/L_3)$?

55. What is $(L_1/L_2)/L_3$?

12.4 Substitution and Homomorphism

Let f be a function that maps each $\sigma \in \Sigma$ to some regular language L_σ. In general, each L_σ may be over its own alphabet Γ_σ distinct from the others, but we can understand all of the L_σ to be languages over $\Gamma = \bigcup_{\sigma\in\Sigma}[\Gamma_\sigma]$. So while the function may map symbols in Σ to languages over some other alphabet Γ, we can take the range of f to be languages over that single alphabet:

$$f : \Sigma \to \boldsymbol{\mathcal{P}}(\Gamma).$$

f is *a substitution of regular languages for* Σ.

Definition 47 *If* $w \in \Sigma^*$ *then*

$$f(w) \overset{\text{def}}{=} \begin{cases} \{\varepsilon\} & \textit{if } w = \varepsilon, \\ f(w') \cdot f(\sigma) & \textit{if } w = w' \cdot \sigma. \end{cases}$$

Definition 48 *If* $L \subseteq \Sigma^*$ *then*

$$f(L) \overset{\text{def}}{=} \{f(w) \mid w \in L\}.$$

Note that if $f(\sigma)$ includes ε then f may erase 'σ's occurring in strings in L, while if $f(\sigma) = \emptyset$ then f has the effect of deleting every string in L in which 'σ' occurs.

Example: Let $f(a) = L(a^+ca^+)$ and $f(b) = (b^+cb^+)$. Let $L = ((ab + ba)^*)$. Then $abab \in L$ and

$$\underbrace{acaaa}_{\in f(a)}\underbrace{bbbcb}_{\in f(b)}\underbrace{aaacaa}_{\in f(a)}\underbrace{bcbbb}_{\in f(b)} \in f(w).$$

And

$$\begin{aligned} f(L) &= f(L((ab+ba)^*)) \\ &= (f(\{ab\} \cup \{ba\}))^* \\ &= (f(\{ab\}) \cup f(\{ba\}))^* \\ &= (f(a) \cdot f(b) \cup f(b) \cdot f(a))^* \\ &= (L(a^+ca^+)L(b^+cb^+) \cup L(b^+cb^+)L(a^+ca^+))^* \\ &= L((a^+ca^+b^+cb^+ + b^+cb^+a^+ca^+)^*). \end{aligned}$$

Theorem 7 *The class of regular languages is closed under substitution by regular languages.*

Proof (sketch)**:** If L is regular then $L = L(r)$ for some regular expression r. Then $f(L) = L(f(r))$ where $f(r)$ is, in essence, the result of applying f to r. $\dashv$

56. Let $L = L(((a + ba)^*ba)^*)$ and $f = \{a \mapsto L(ab^*a), b \mapsto L(b^*ab^*)\}$. Give a regular expression for the language $f(L)$.

Definition 49 *Let $h : \Sigma \to \Gamma^*$ map symbols of Σ to strings over some alphabet Γ. We say that h is a* homomorphism *of Σ to Γ^*, the* homomorphic image *of $w \in \Sigma^*$ (under h) is*

$$h(w) \stackrel{\text{def}}{=} \begin{cases} \varepsilon & \textit{if } w = \varepsilon \\ h(w') \cdot h(\sigma) & \textit{if } w = w' \cdot \sigma \end{cases}$$

and the homomorphic image *of $L \subseteq \Sigma^*$ (under h) is*

$$h(L) \stackrel{\text{def}}{=} \{h(w) \mid w \in L\}.$$

Corollary 5 *The class of regular languages is closed under homomorphisms.*

This is because a homomorphism is, in essence, a substitution in which $\mathbf{card}(f(\sigma)) = 1$ for all $\sigma \in \Sigma$. Thus closure under substitution implies closure under homomorphism.

12.5 Reversal

Theorem 8 *The class of regular languages is closed under reversal.*

Proof (sketch)**:** Let $L = L(r)$ for some regular expression r. Let

$$r^{\mathrm{R}} \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } r = \emptyset \\ \varepsilon & \text{if } r = \varepsilon \\ \sigma & \text{if } r = \sigma \\ (s^{\mathrm{R}} + t^{\mathrm{R}}) & \text{if } r = (s + t) \\ (t^{\mathrm{R}} \cdot s^{\mathrm{R}}) & \text{if } r = (s \cdot t) \\ (s^{\mathrm{R}^*}) & \text{if } r = (s^*). \end{cases}$$

It follows by an easy induction on the structure of w that $w \in L(r) \Leftrightarrow w^{\mathrm{R}} \in L(r^{\mathrm{R}})$. Thus

$$L^{\mathrm{R}} = L(r)^{\mathrm{R}} = L(r^{\mathrm{R}}).$$

$\dashv$

Proof (sketch, alternate)**:** Let $L = L(\mathcal{A})$ for some DFA $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$. Let $\mathcal{A}' = \langle Q', \Sigma, \delta', q_0', F'\rangle$ be an *NFA* with:

$$\begin{aligned} Q' &= Q \cup \{q_0'\} \\ \delta'(q_0', \varepsilon) &= F \\ \delta'(q, \sigma) &= \{p \mid \delta(p, \sigma) = q\} \\ F' &= \{q_0\}. \end{aligned}$$

So $\mathcal{A}'$ is, in essence, the NFA one gets by reversing the edges of $\mathcal{A}$.

57. Why is $\mathcal{A}'$ an NFA rather than another DFA.

58. Complete the proof by showing that

$$w \in L(\mathcal{A}) \Leftrightarrow w^{\mathrm{R}} \in L(\mathcal{A}').$$

[**Hint:** Prove first that $\hat{\delta}(q, w) = p \Leftrightarrow q \in \hat{\delta}'(p, w^{\mathrm{R}})$ for all $q, p \in Q$ (that is, every state but q_0').]

⊣

59. Prove that the class of regular languages is closed under Suffix.
[**Hint:** Why is this question here rather than in Section 12.3?]

12.6 Using Closure Results to Prove Languages are Non-regular

Let $L_{ab} = \{a^i b^i \mid i \geq 0\}$. We will take L_{ab} to be our *canonical* non-regular language. We can then use the known closure results for the class of regular languages to prove, by contradiction, that some language L is not regular by showing how to reduce L to L_{ab} using operations that preserve regularity.

Example: Let $L = \{w \in \{a, b, c, d\}^* \mid |w|_{ab} \geq |w|_{cd}\}$. To show that L is not regular.
Let $L_1 = L \cap L((ab)^*(cd)^*)$. Then

$$L_1 = \{(ab)^i (cd)^j \mid 0 \leq j \leq i\}.$$

Let $L_2 = h_1(L_1)$ where $h_1 = \{a \mapsto a, b \mapsto \varepsilon, c \mapsto b, d \mapsto \varepsilon\}$. Then

$$L_2 = \{a^i b^j \mid 0 \leq j \leq i\}.$$

Let $L_3 = h_2(L_1)$ where $h_2 = \{a \mapsto b, b \mapsto \varepsilon, c \mapsto a, d \mapsto \varepsilon\}$. Then

$$L_2 = \{b^k a^l \mid 0 \leq l \leq k\}.$$

Then

$$L_2 \cap {L_3}^{\mathrm{R}} = \{a^i b^j \mid 0 \leq j \leq i\} \cap \{a^l b^k \mid 0 \leq l \leq k\} = \{a^i b^j \mid 0 \leq j = i\} = L_{ab}.$$

As the class of regular languages is closed under intersection, homomorphism and reversal if L were regular L_{ab} would be regular as well. But L_{ab} is our canonical *non-regular* language and, consequently, L cannot be regular.

60. Consider again a system of two processes (A and B) exchanging messages as in Exercise 25. Again A sends either 'm_1' or 'm_2' and B acknowledges with 'a_1', 'a_2' or 'a_{12}', where 'a_1' acknowledges 'm_1', 'a_2' acknowledges 'm_2' and 'a_{12}' acknowledges both. In contrast to Exercise 25, we will now allow any number of 'm_1's or 'm_2's to be outstanding. We require only that every message is eventually acknowledged and that no acknowledgment is sent unless there is some outstanding message(s) of the corresponding type. Show that the set of finite sequences of messages that satisfy this protocol is *not* regular.
[**Hint:** Start by taking an intersection with a regular set to simplify the language. (Get rid of all the 'm_2's, 'a_2's, and 'a_{12}'s.)]

13 Some Decision Problems for the Class of Regular Languages

We'll close this study of the regular languages by considering whether certain questions concerning given regular languages can be decided algorithmically. We will focus on a few questions, with you doing a couple more as exercises.

Membership: Given a string and a finite representation of regular language, is the string in the language?

Emptiness: Given a finite representation of a regular language, is that language empty?

Finiteness: Given a finite representation of a regular language, is that language finite?

Equivalence: Given finite representations of two regular languages, do they represent the same language?

We will generally assume that the representation used in the instances of these problems are DFAs. This is usually the easiest form to handle and, as we have already established that there are algorithms for translating other representations into the form of DFAs, if the problem can be decided given DFAs it can be decided given any other representation.

61. Why don't the instances of these problems just include the languages themselves rather than representations of the languages?

62. Which of these properties are algorithmically decidable for the class of finite languages?

These problems are all of the type we called "checking problems" in Section I. They are more properly known as *decision problems*: given some instance decide if it satisfies some property. If such a problem can be solved algorithmically the corresponding property is said to be *decidable.*

13.1 Membership

Given a string and a DFA, is the string in the language accepted by the DFA?

The question here is whether the computation of the DFA on the string terminates in an accepting state. The obvious way of approaching this is to simply simulate the DFA: start with the initial ID, calculate its successor (using the transition function), repeat until a terminal ID is reached, and answer yes iff the terminal ID includes an accepting state. This is an effective procedure—each step can actually be carried out—and it will certainly give the right answer when it finishes. The issue we need to address is whether it will always finish—is it an algorithm? Here we can appeal to our initial discussion of computations in Section 7. If an ID has a successor the length of the remaining input in that successor is exactly one less than the length of the remaining input in the ID. Thus, there are exactly $|w|$ successors in the computation of any DFA on w.

63. Which is to say, the length of the computation in transitions (steps) of the DFA is $|w|$. What is the length of the computation in terms of its representation as a sequence of IDs; how many IDs are in the sequence?

Consequently, in simulating the DFA, the process of computing the successor will be repeated exactly $|w|$ times. Since strings have finite length, we are guaranteed to reach a terminal ID in a finite number of steps.

Theorem 9 *Membership is decidable for the class of regular languages.*

It is useful to consider this approach from the perspective of transition graphs as well. In exploring the computation of the DFA on w we are simply following the path labeled w in the transition graph of the DFA that starts at q_0. For DFAs there is only one such path and it consists of exactly $|w|$ edges.

64. Can we establish such a bound on the computations of NFAs without ε-transitions? With them? On paths in transition graphs?

13.2 Emptiness

Given a DFA, is the language accepted by that DFA empty? Membership asks us to decide whether there is an accepting computation on a given input. Emptiness asks us to decide whether there is a accepting computation on *any* input. Since there are infinitely many strings that might be accepted, this is, in general, more difficult: there are systems of computation for which membership (or its equivalent) is decidable but emptiness is not. While we might approach this by applying our algorithm for membership systematically to all strings over the alphabet of the DFA—starting with the empty string, say, and then all strings of length one, then two, etc.—we cannot check all such strings in finitely much time. For this approach to work we need to identify a finite subset of the strings that suffices: a set we *can* check exhaustively which is guaranteed to include some string in the language if the language includes any string.

We can identify such a subset by thinking back to the Pumping Lemma (Section 10, Lemma 13). This says that there is number n that depends only on the DFA, such that if some string x with length n or more is in the language accepted by the DFA then there is some string shorter than x in the language (the string in which v is pumped zero times). Moreover, the length of that string is no less than $|x| - n$ (since, $|uv| \leq n$, and *a fortiori* $|v| \leq n$).

Suppose, then, that there is some string of length n or more in the language. Let w_0 be such a string with minimal length, i.e. w_0 is in the language, $|w_0| \geq n$ and every string with length n or more that is in the language is at least as long as w_0. How long is w_0 with v pumped zero times? Since this is strictly shorter than w_0 and is in the language, and, by choice of w_0, every string in the language shorter than w_0 is shorter than n, it must be the case that w_0 with v pumped zero times is shorter than n. Thus, we can limit our search to strings of length strictly less than n.

All that remains is to figure out what n is for the given DFA. In proving the pumping lemma we used a pigeon hole principle argument to show that the computation of a DFA on any string longer than the number of states (that is $\mathbf{card}(Q)$) must include a loop. Cutting out this loop is what gave us the accepting computation of the DFA on the string with v pumped zero times. Thus, n can be taken to be equal to $\mathbf{card}(Q)$.

The algorithm, then, consists of applying the membership algorithm to all strings over the alphabet of the DFA that are no longer than the size of its state set. If any of these strings are in the language they witness the fact that it is non-empty. If, on the other hand, none of them are, we know as a consequence of the pumping lemma that no longer string is in the language either.

This is even simpler if we think in terms of the transition graph. In that context the emptiness problem is simply asking if there is any path in the graph from the start state to an accepting state. Algorithms for solving this problem (on finite graphs) should be well-known to you (e.g., Dijkstra's or Floyd's algorithms). One of the attractions of representing DFAs as transition graphs is the fact that known graph algorithms can be employed to solve their decision problems.

13.3 Finiteness

Given a DFA, is the language accepted by that DFA finite? Just as the emptiness problem can be seen as a (potentially more difficult) generalization of the membership problem, the finiteness problem is, in a particular sense, a generalization of the emptiness problem. Here we need to determine not only if *any* string is in the language accepted by the DFA but *how many* of them there are, in particular, if there are only finitely many of them.

Thinking, again, in terms of the pumping lemma, if there are any strings in the language of length n or greater then there will be infinitely many of them (since we can pump v any number of times). Conversely, if there is no string in the language of length greater than n then there are but finitely many strings in the language (since the number of strings over a given alphabet of length less than n is finite). So again, we can use the membership algorithm, now searching for strings of length n or greater. And again, our problem is to establish an upper bound on the length of the strings we test.

Consider, again, w_0, a string of minimal length among those of length n or greater in the language. How long is w_0? We have established that, by choice of w_0, the length of w_0 with v pumped zero times is strictly less than n, and, by the hypothesis of the pumping lemma, the length of v is no greater than n, we can simply calculate that $n \leq |w_0| < 2n$. Thus we need only search for some string in the language of length between n and $2n$.

65. Give an algorithm for deciding finiteness that is based on known algorithms for deciding problems for graphs.

13.4 Equivalence

Given two DFAs, do they accept the same language? Here, again, we have a sort of generalization of the emptiness problem. We need not only to establish whether there is any string in the language accepted by a DFA, but whether the set of such strings for one DFA is the same as those accepted by another. In this case the pumping lemma is not much help. Instead, we will appeal to the Myhill-Nerode Theorem (Section 11) and, in particular, the result of Lemma 18 (Section 11.3). This tells us that the result of minimizing a DFA using the construction of Section 11.3 is unique up to isomorphism, that is to say, is identical to all other minimal size DFAs accepting the same language except, possibly, for the actual names of the states. Isomorphism of edge-labeled graphs is another problem for which an algorithm is known (although perhaps not as familiar). We can solve equivalence of DFAs, then, by minimizing them and using the graph algorithm to test isomorphism.

66. Sketch an algorithm to decide isomorphism of DFAs.

We can establish decidability of emptiness even more easily if we combine the closure results of the previous section with earlier results of this section.

67. Suppose $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$. What is $L(\mathcal{A}_1) \setminus L(\mathcal{A}_2)$?

68. Suppose $L(\mathcal{A}_2) \subseteq L(\mathcal{A}_1)$. What is $L(\mathcal{A}_2) \setminus L(\mathcal{A}_1)$?

69. Show that there is an effective construction that, given DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, builds a DFA accepting $L(\mathcal{A}_1) \setminus L(\mathcal{A}_2) \cup L(\mathcal{A}_2) \setminus L(\mathcal{A}_1)$. (This is known as the *symmetric difference* of the languages.) You do not need to give the actual construction, simply show how the constructions of Section 12 can be combined to make such a construction.

70. Use this result, along with decidability of emptiness, to show that equivalence of DFAs is decidable.

We close with a couple of exercises.

71. In Section 12 we established that the *class* of regular languages was closed under reversal: L regular implies L^{R} regular. Let us say that a *language* L is closed under reversal iff $w \in L$ implies $w^{\mathrm{R}} \in L$. Prove that the question of whether a given regular language is closed under reversal is decidable.
[**Hint:** Use the closure properties and decision procedures we have already established.]

72. Show that decidability of both emptiness and membership is a consequence of decidability of equivalence, i.e., show how an algorithm for equivalence can be used (as a subroutine) to build an algorithm for emptiness or an algorithm for membership.
[**Hint:** For emptiness start out by giving a DFA that accepts the empty language. For membership start out by sketching and algorithm that, given w, constructs a DFA accepting $\{w\}$.]