Chapter 2

Finite Automata

This chapter and much of the text concerns strings, which are always defined with respect to a specific alphabet. Recall that an alphabet is a set of characters. For example:

- Binary = \{0,1\}
- Boolean = \{T,F\}
- English = \{a, b, \cdots, z\}

The usual way of writing an unspecified $k$-character alphabet is as \( \Sigma = \{ a_1, a_2, \cdots, a_k \} \).

2.1 Regular Expressions: A Motivating Problem

String matching is a common task in text processing. In string matching there are two inputs, a text \( t \) and a pattern \( p \), which are both strings of characters over the same alphabet. The task is to find all occurrences of \( p \) in \( t \); for example, to find all occurrences of the string "Big Bang Theory" (the \( p \)) in an astronomy textbook (the \( t \)). A simple algorithm would be to check for an occurrence of \( p \) starting at each successive location in \( t \). It is easy to implement this to run in time \( O(|p| \cdot |t|) \). There are also several well-known linear time algorithms for this problem, i.e. running in time \( O(|p| + |t|) \), most notably the Knuth-Morris-Pratt and the Boyer-Moore string matching algorithms; however, these lie outside the scope of this text.

Often, one would like to perform more complicated types of searches. For example, one might want to search for the singular or plural form of a word, and sometimes this is not just a matter of adding an "s", as with "mouse" and "mice"; of course, in this case, one could just perform two searches, one for each word. But, suppose one wants to search for every phone number in a text. One might describe this as consisting of any sequence of 10 digits in the range 1–9 (at least for US phone numbers), or one might give a more elaborate collection of patterns, which include parentheses, spaces and dashes, as in numbers of the form (123) 456-7890 and 123 456 7890, etc. Presumably, one is not going to search one by one for all 10^{10} or more possible patterns.

More generally, one can describe a class of strings that are specified as regular expressions (to be defined shortly). Interestingly, several Unix commands, such as grep, allow one to search in text files for sets of strings specified by regular expressions. Furthermore, these searches can be done efficiently. One of the questions we will be answering in this chapter is how this is possible.
Regular expressions allow one to specify sets (languages) built up by means of the following three operations: union, concatenation, and iteration (defined below). These are called the Regular Operations.

- Union: \( A \cup B = \{ w \mid w \in A \text{ or } w \in B \} \).
- Concatenation: \( A \circ B = \{ uv \mid u \in A \text{ and } v \in B \} \). This is the set of strings comprising a string from \( A \) followed by a string from \( B \).
- Iteration: \( A^* = \{ x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and } x_i \in A, \text{ for } i = 1, 2, \ldots, k \} \). Note that setting \( k = 0 \) shows \( \lambda \in A^* \). This is the set of strings consisting of the concatenation of zero or more strings from \( A \). One can also view this as starting with the empty string \( \lambda \), and concatenating 0 or more strings from \( A \) to the initial \( \lambda \). This is often called the star operation, for obvious reasons.

Regular expressions are simply a notation to represent the above operations. We will use small letters such as \( r, s, r_1 \), etc., to denote regular expressions. Each regular expression specifies a set or language of strings; we also say that the regular expression represents the corresponding language. Also, note that each regular expression is defined with respect to a specified alphabet \( \Sigma \).

Regular expressions are built using the following six rules, comprising three base cases and three combining rules. (This can be thought of as being analogous to the building of arithmetic expressions using the operators \( "+", "-", "\ast", \) and \( "/" \), along with parentheses.)

We begin with the base cases.

1. \( \phi \) is a regular expression; it represents \( \phi \), the empty set.
2. \( \lambda \) is a regular expression; it represents the language \( \{ \lambda \} \), the language containing the empty string alone.
3. \( a \) is a regular expression; it represents the language \( \{ a \} \).

For the combining rules, let regular expressions \( r \) and \( s \) represent languages \( R \) and \( S \), respectively. The rules follow.

4. \( r \cup s \) is a regular expression; it represents language \( R \cup S \).
5. \( r \circ s \) is a regular expression; it represents language \( R \circ S \). We write \( rs \) for short.
6. \( r^* \) is a regular expression; it represents language \( R^* \).

Parentheses are used to indicate the scope of an operator. It is also convenient to introduce the notation \( r^+ \); while not a standard regular expression, it is shorthand for \( rr^* \), which represents the language \( RR^* \). Also, \( \Sigma \) is used as a shorthand for \( a_1 \cup a_2 \cup \cdots \cup a_k \), where \( \Sigma = \{ a_1, a_2, \ldots, a_k \} \). Sometimes, to emphasize that \( r \) represents a language, we write \( L(r) \) to denote the language represented by regular expression \( r \). Finally, if \( w \in L(r) \), \( w \) is said to be represented by \( r \).

Examples. Let \( \Sigma = \{ a, b \} \).

1. \( a^*ba^* = \{ w \mid w \text{ contains exactly one } b \} \). This can be read as zero or more \( a \)'s, followed by a \( b \), followed by zero or more \( a \)'s.
2. $\Sigma^*bab\Sigma^* = \{w \mid w \text{ contains } bab \text{ as a substring}\}$. This can be read as a possibly empty string, followed by the string $aba$, followed by a possibly empty string.

3. $(\Sigma \Sigma)^* = \{w \mid w \text{ has even length}\}$. This can be read as a sequence of zero or more pairs of characters.

4. $a\Sigma^*a \cup b\Sigma^*b \cup a \cup b = \{w \mid w \text{ has the same first and last character}\}$.

5. $(a \cup \lambda)b^* = ab^* \cup b^*$.

6. $a^*\phi = \phi$. (Why?\footnote{As there is no string in $\phi$, to follow a string in $a^*$ with a string from $\phi$ is not possible.})

One might ask why this extra notation has been introduced; why not simply stick with the set notation? The answer is that one can be terser, for example writing $a^*ba^*$ rather than $\{a\}^* \circ \{b\} \circ \{a\}^*$.

Now, given a regular expression $r$, which might represent an infinite set of strings, and given a text $t$, how can one find every substring $s$ of $t$ such that $s$ lies in the set of strings specified by $r$, let alone do this efficiently? Some instances may seem quite doable; for example searching for $aa^*$, the set of strings containing one or more $a$’s, seems quite feasible. But handling an arbitrary expression involving multiple concatenations, unions, and iterations (or stars) seems more daunting.

To answer this question we are going to start by looking at a class of devices or machines called finite automata, which as first sight will seem quite unrelated to this problem.

### 2.2 Finite Automata: The Basic Model

Finite automata model very simple computational devices or processes. These devices have a constant amount of memory and process their input in an online manner. By that we mean that the input is read a character at a time, and for each character, as it is read, a constant amount of processing is performed.

One might wonder whether such simple devices could be useful. In fact, they are widely used as controllers in mechanical devices such as elevators and automatic doors.

**Example.** A controller for the doors on the exit hatchway of a spaceship.

The rather simple spaceship in this example has an exit area with two doors: one leads out into space, the other leads into the spaceship interior, as shown in Figure 2.1.

![Spaceship Exit Doors](image)

Figure 2.1: Spaceship Exit Doors
is simply a directed graph with labels on the edges.

![Spaceship Door Controller Diagram](image)

**Figure 2.2: Spaceship Door Controller**

The vertex names in this graph specify the door state to which the vertex corresponds. The edges exiting a vertex indicate the actions taken by the controller on receiving the request labeling the edge. So for example, if both doors are closed and a request to open the exit door is received, then the controller opens this door and the state changes accordingly. On the other hand, if currently the interior door is open and a request to open the exit door is made, then nothing changes; i.e. the interior door remains open and the exit door remains closed. Note that there are many requests that result in no change. We can simplify the diagram so as to show only those requests that cause a change of state, as shown in Figure 2.3.

![Sparse Diagram](image)

**Figure 2.3: Spaceship Door Controller, Sparse Diagram**

As requests are received the controller cycles through the appropriate states. To implement the device it suffices to have a 2-bit memory to record the current state (2 bits can distinguish 4 states) plus a little logic to decide what state change to make as requests are received.

**Comment.** The same design of controller would be needed for the doors on a torpedo hatchway
2.2. FINITE AUTOMATA: THE BASIC MODEL

in a submarine.

Many controllers need similarly modest amounts of memory and logic. They are more readily understood as simple devices, specified by means of a graph, rather than as general purpose computers.

Similar devices are also very useful as a front end in many programs, where they are used to partition the input into meaningful tokens. For example, a compiler, which is a program that processes input programs so they can then be executed, needs to partition the input into keywords, variables, numbers, operators, comments, etc.

For example, we could imagine recognizing the keywords if, then, else, end with the help of the following device or procedure, represented as a graph.

The automata always has at most one active vertex. Initially, the active vertex is the start vertex (the vertex labeled “λ read” in this example). On reading the next input character, the automata traverses the edge, if any, exiting the active vertex and labeled by this character; the vertex reached becomes the new active vertex. In other words, the automata follows the path whose concatenated edge labels form the string s. If at some point there is no edge to follow, then the computation fails; otherwise, if the active vertex when the computation finishes is a vertex represented by a double circle, then the computation has recognized a keyword; if is finishes at some other vertex, then the string read is not a keyword. For example, on reading the input string if, the finite automata goes from the start vertex to the vertex labeled if. Note that it is conventional for the start vertex, which need not be named start, to be indicated in the graph by means of a double arrow.

There are more details to handle in a compiler. In particular, the compiler has to be able to decide when it has reached the end of a keyword rather than being in the middle of a variable (e.g. ended) whose name begins with the keyword. But these essentially straightforward and tedious, albeit important, details are not the focus here.

Variables are another collection of strings that compilers need to recognize. Let us suppose variables consist of all strings of (lower-case) letters other than keywords, and to simplify the illustration below let us suppose there is just one keyword: if. Then the following procedure, shown in Figure 2.5, will recognize variables.

Notice that the graph has a self-loop; also, some of its edges have more than one label (alternatively, we could think of this as a collection of parallel edges, each with a distinct label). Again, to use the variable recognizer, the compiler follows the path spelled out by the input. Any input other than the word if leads to a vertex represented as a double circle and hence represents a variable.

2.2.1 A Precise Specification

We need to introduce a little notation and terminology.

By convention, a finite automaton is named \( M \), or \( M_1, M_2, \ldots \) if there are several. As we shall see subsequently, other devices will also be named by \( M \). A finite automaton is a directed graph
plus a finite alphabet $\Sigma$ in which:

- One vertex is designated as the start vertex.
- A subset of the vertices, conventionally called $F$, form the collection of Recognizing or Final vertices.
- Each edge is labeled by a character of $\Sigma$.
- Each vertex has $|\Sigma|$ outgoing edges, one for each character in $\Sigma$.

In the corresponding drawing of the automaton, the start vertex is indicated by a double arrow, and the vertices in $F$ are indicated by drawing them with double circles.

One way to think about $M$’s processing is procedurally. At any point in time, $M$ will have one currently reached vertex, or current vertex for short. At the start of the computation, before any of its input $u$ has been read, the current vertex is the start vertex. In general, having read a portion $v$ of its input, the current vertex is the vertex $p$ it reaches on reading string $u$. $M$ then reads the next character of its input, $a$ say, and follows the edge labeled $a$ leaving vertex $p$, which brings it to vertex $q$. $q$ is now its current vertex. Given an input string $s$, $M$ will read $s$ character by character, advancing from vertex to vertex as described above, until all of $s$ is read, which brings it to some last vertex $r$, say. Then $s$ is recognized exactly if $r$ is in the recognizing or final set of vertices, $F$. We call $r$ the destination vertex for string $s$, i.e. the vertex the automata reaches after reading all of $s$.

We can also think of this computation as causing $M$ to follow a path $P$, where the concatenation of the edge labels on $P$ form the string $s$. We think of $s$ as $P$’s label; then $M$’s computation can be viewed as finding the end vertex of the path starting at the start vertex and labeled by $s$. For example, in the Keyword Recognizer (see Figure 2.4), the input $if$ specifies the path from the vertex start to the vertex named $if$. If the end of the path is a vertex in $F$ then $M$ recognizes or accepts $s$; otherwise $M$ rejects $s$. Continuing the example, input $if$ is recognized by the automaton as the vertex $if$ is in $F$.

Thus we can view $M$ as partitioning the set of all possible strings into those it recognizes and those it rejects. The set of strings recognized by $M$ is often called the language recognized or
accepted by \( M \), and is sometimes written as \( L(M) \) or \( L \) for short. The topic we will study is what sorts of collections of strings finite automata can recognize.

Conventionally, the vertices of a finite automata are called its states and are written as \( Q = \{ q_1, q_2, \cdots, q_r \} \) rather than \( V = \{ v_1, v_2, \cdots, v_r \} \). The index \( n \) tends to be reserved for the length of the input string. However, we will seek to provide more descriptive specifications for the vertices in our examples.

For simplicity in drawing we replace multiple parallel edges by a single edge with multiple labels as shown in Figure 2.6.

![Figure 2.6: Simplifying Figures of Finite Automata](image)

Also, it is often convenient to omit a “sink” vertex (a non-recognizing vertex which cannot be left); see Figure 2.7.

![Figure 2.7: Omitting the Sink Vertex](image)

Note the descriptive specifications being used for the vertices. A description for a vertex \( v \) specifies the strings \( s \) that cause the automata to reach vertex \( v \) after reading all of \( s \) when starting from its start vertex. For example, in Figure 2.7, the vertex labeled ‘ye’ is the destination on reading the input string ‘ye’, but it is the destination for no other input string.

Finally, it’s helpful to have a notation for representing labeled edges. We write \( \delta(p, a) = q \) to mean that there is an edge labeled \( a \) from vertex \( p \) to vertex \( q \); a more active interpretation is that starting at vertex (state) \( p \), on reading \( a \) the automata goes to vertex \( q \). In other words the destination on reading \( a \) when at vertex \( p \) is vertex \( q \). Conventionally \( \delta \) is called the transition function. We will also refer to the labeled edge as a transition rule, because an edge \( (p, q) \) labeled by an \( a \) tells the finite automata what to do if the character \( a \) is read when it is at vertex \( p \): namely to transit to vertex \( q \).
CHAPTER 2. FINITE AUTOMATA

It is also useful to extend the destination (or transition) function to the reading of strings; this is denoted by \( \delta^* \). \( \delta^*(p, s) = r \) means that vertex \( r \) is the destination on reading string \( s \) starting at vertex \( p \). For example, in the Variable Recognizer (see Figure 2.5), \( \delta^*(\lambda \text{ read}, \text{it}) = \text{“all other variable names”}^2 \). Observe that \( \delta^*(p, \lambda) = p \): reading the empty string, i.e. reading no characters, leaves the currently reached vertex unchanged.

**Mathematical notation.** A finite automata \( M \) can be written as \( M = (Q, \Sigma, \delta, \text{start}, F) \), where \( Q \) is its set of vertices, \( \text{start} \in Q \) is its start vertex, \( F \subseteq Q \) is its set of recognizing vertices, \( \Sigma \) is the input alphabet, and \( \delta \) specifies its edges and their labels.

**Some Particular Languages**

- \( L(M) = \phi \) — This is the empty language, the language with no strings in it. So the finite automata \( M \) recognizes nothing.

- \( L(M) = \{\lambda\} \) — This is the language comprising the single string \( \lambda \). Note that \( \phi \neq \{\lambda\} \); these two languages are distinct.

### 2.2.2 Examples of Automata

\( M_1 \), shown in Figure 2.8, recognizes the following strings: those strings with at least one \( a \) and an even number of \( b \)'s following the last \( a \).

\[
\begin{array}{c}
\text{no } a \\
\longrightarrow \text{ even number of } b \text{'s after last } a \\
\text{odd number of } b \text{'s after last } a \\
\end{array}
\]

Figure 2.8: Example Automata \( M_1 \)

\( M_2 \), shown in Figure 2.9, recognizes the set \( L(M_2) = \{x \mid x \in \{a, b\}^* \text{ and } x \text{ ends with an } a \} \).

\[
\begin{array}{c}
x = \lambda \text{ or } \\
\text{x ends with } \\
a b \\
\end{array}
\]

Figure 2.9: Example Automata \( M_2 \)

\( M_3 \), shown in Figure 2.10, recognizes the complement of the language recognized by \( M_2 \).

\( M_4 \), shown in Figure 2.11, keeps a running count mod 3 of the sum of the characters in its input that it has read, where the input alphabet \( \Sigma = \{0, 1, 2\} \); it recognizes a string if the corresponding total is 0 mod 3.

---

2 When a name or string includes spaces we will often put quote marks around it to avoid ambiguity.
2.2. FINITE AUTOMATA: THE BASIC MODEL

Figure 2.10: Example Automata $M_3$: $L(M_3) = \{ x \mid x$ is the empty string or $x$ ends with a $b$ \}

Suppose that we want to maintain the above running count, but mod $k$, where the input alphabet is $\Sigma = \{0, 1, \cdots, k - 1\}$. This can be done with a $k$-vertex automata. The automata has vertex set $Q = \{q_0, q_1, \cdots, q_{k-1}\}$; $q_i$ will be its currently reached vertex if the running sum equals $i \mod k$. It immediately follows that the start vertex is vertex $q_0$ and the recognizing vertex set is $F = \{q_0\}$. The transition function is also immediate: if the current running sum equals $h \mod k$ and an $i$ is the next input character read, then the new running sum is $j = h + i \mod k$; so the current vertex changes from $q_h$ to $q_j$. We write $\delta(q_h, i) = q_j$ where $j = h + i \mod k$. (See Figure 2.12.)

Figure 2.11: Example Automata $M_4$: $L(M_4) = \{ x \mid \sum_{i=1}^{\lfloor x \rfloor} x_i = 0 \mod 3$, where $x = x_1x_2\cdots x_{\lfloor x \rfloor} \}$. Note that $\lambda \in L(M_4)$.

Figure 2.12: Example Automata $M_5$: The Transition Function
2.2.3 Designing Automata

The first issue is to decide what has to be remembered as the input is read. The easiest thing, it might seem, is to remember the whole string. But, in general, one cannot do this as the automata has only finite memory.

For example, with $\Sigma = \{a, b\}$, consider recognizing those strings with an odd number of $b$’s. It suffices to keep track of whether an even or an odd number of $b$’s have been read so far. This leads to the transition diagram shown in Figure 2.13(a). To obtain the full automata, it’s enough to note that at the start zero $b$’s have been read (an even number), and the recognizing vertex corresponds to an odd number of $b$’s having been read; see Figure 2.13(b).

![Figure 2.13: Automata recognizing strings with an odd number of b’s](image)

Next, consider the language $L(M) = \{w \mid w \text{ contains at least one } a\}$, with $\Sigma = \{a, b\}$. The natural vertices correspond to “no $a$ read” and “at least one $a$ read”. The transitions are now evident and are shown in Figure 2.14.

![Figure 2.14: Automata recognizing strings with at least one a](image)

Finally, consider the language $L(M) = \{w \mid w \text{ contains at least two } b\text{'s}\}$, with $\Sigma = \{a, b\}$. The natural three vertices correspond to: “no $b$ read”, “one $b$ read”, and “at least two $b$’s read”. Again, the transitions are evident and are shown in Figure 2.15.

The vertex descriptors help to check that the automata is doing what it is supposed to do. In particular, when designing an automata, you should check that the edge labels and the descriptors are consistent.

More specifically, suppose that $(p, q)$ is an edge with label $a$. Let $S_p$ be $p$’s descriptor and let $S_q$ be $q$’s. e.g. in Figure 2.8, the start vertex descriptor is the set of all strings with no $a$’s, i.e. $\{\lambda, b, bb, bbb, \ldots\}$. Then you want to confirm that for each string $s \in S_p$, string $sa \in S_q$; i.e. appending an $a$ to a string in $S_p$ always yields a string in $S_q$. 
2.2. FINITE AUTOMATA: THE BASIC MODEL

This may seem very difficult to do, but in fact it tends to be straightforward. For example, consider Figure 2.8 again, and the edge (start, even) labeled $a$. We need to confirm that the descriptor 'even number of $b$'s after last $a$' includes all strings of the form 'no $a$' (strings in $S_{start}$) followed by one $a$. Clearly, there are zero $b$'s after the last (just read) $a$, and zero is an even number of $b$'s. Of course it would be rather tedious to write out many such checks; one just wants to do them in one's head. However, it is all too easy to assume one has got the design right and not bother with the checks; this is how many simple mistakes are overlooked.

There are three other very simple checks.

1. When nothing has been read, the automata is at the start vertex. Consequently, the empty string must be in the descriptor for the start vertex.

2. Each string causes exactly one vertex to be reached. Consequently, the descriptors for distinct vertices are disjoint. In addition, the union of the descriptors is the set of all strings.

3. The union of the sets of strings specified by the descriptors for the set of recognizing vertices specifies exactly those strings that you want the automata to recognize. More formally, let $F = \{q_1, q_2, \ldots, q_i\}$; then $S_{q_1} \cup S_{q_2} \cup \cdots \cup S_{q_i}$ is the set of strings recognized by the automata.

In Section 2.6 we will show that performing the full set of checks ensures the correctness of the automata.

2.2.4 Recognizing More Complex Sets

We consider sets (languages) built up by means of the following three operations: union, concatenation, and star. Recall that these are called the Regular Operations (see Section 2.1). As we will see, if the sets $A$ and $B$ can be recognized by finite automata, then so can their union, concatenation, and the "star" of each set.

**Recognizing $A \cup B$** The approach is natural: we use a device $M_{A\cup B}$ that comprises two finite automata: one, $M_A$, that recognizes $A$, and one, $M_B$, that recognizes $B$. The device $M_{A\cup B}$ runs $M_A$ and $M_B$ in parallel (simultaneously) on its input and recognizes the input exactly if, on reading this input, the destination in at least one of $M_A$ and $M_B$ is a recognizing vertex (state).

The only challenge is to show how to have $M_{A\cup B}$ be a finite automata. To describe this we need some more notation. Let $M_A = (Q_A, \Sigma, \delta_A, \text{start}_A, F_A)$, $M_B = (Q_B, \Sigma, \delta_B, \text{start}_B, F_B)$, and $M_{A\cup B} = (Q_{A\cup B}, \Sigma, \delta_{A\cup B}, \text{start}_{A\cup B}, F_{A\cup B})$. We illustrate the construction in Figure 2.16 for the pair of languages $A = \{\text{even length strings}\}$ and $B = \{w \mid \text{all } a's \text{ in } w \text{ precede all the } b's \text{ in } w\}$.
Consider what happens on input \textit{aaba}. \(M_A\) and \(M_B\) go through the following sequence of pairs of vertices (the first vertex in the pair is \(M_A\)'s vertex, the second is \(M_B\)'s: \((p_1, q_1), (p_2, q_1), (p_1, q_1), (p_2, q_2), (p_1, q_3)\). Clearly, the information \(M_{A\cup B}\) needs to record is the current pair of vertices reached by \(M_A\) and \(M_B\), respectively. Thus, for this example, \(M_{A\cup B}\) will need to have 6 vertices, one vertex for each possible pair \((p_i, q_j)\). The edges in \(M_{A\cup B}\) need to correspond to the joint moves made by \(M_A\) and \(M_B\). For example, there is an edge labeled \(a\) from \(p_2\) to \(p_1\), and an edge labeled \(a\) from \(q_2\) to \(q_3\); thus in \(M_{A\cup B}\) there will be an edge labeled \(a\) from \((p_2, q_2)\) to \((p_1, q_3)\). The remaining edges are obtained similarly, as shown in Figure 2.16. Clearly, the start vertex of \(M_{A\cup B}\) is the vertex which corresponds to the pair of start vertices in \(M_A\) and \(M_B\), respectively, namely, \((p_1, q_1)\). To see how to identify the recognizing vertices in \(M_{A\cup B}\) we argue as follows. A string \(s\) should be recognized by \(M_{A\cup B}\) exactly if \(s \in A\) or \(s \in B\) (or both). The first of these occurs if the destination vertex \(p\) in \(M_A\) on input \(s\) is in \(F_A\), and the second occurs if the destination vertex \(q\) in \(M_B\) on input \(s\) is in \(F_B\). In other words, the (vertex of \(M_{A\cup B}\) corresponding to the) vertex pair \((p, q)\) needs to satisfy the condition that either \(p \in F_A\) or \(q \in F_B\) (or both). In our example, we see that every vertex pair apart from \((p_2, q_3)\) satisfies this condition.

![Diagram](image)

Figure 2.16: Automata \(M_{A\cup B}\)

Notice that in our example, if we overlay the columns this yields automata \(M_A\); in other words, looking at just the first component of \(M_{A\cup B}\)'s vertex pairs on input \(w\) will yield the same actions as for \(M_A\) on input \(w\). Similarly, overlaying the rows yields automata \(M_B\).

Now we turn to the construction for general regular languages \(A, B\). The vertices of \(M_{A\cup B}\) are
pairs, where each pair consists of a vertex of $M_A$ and a vertex of $M_B$; we call these vertex-pairs for clarity. The purpose of the $M_A$ component in the vertex-pair is to simulate the computation of $M_A$, and similarly that of the $M_B$ component is to simulate $M_B$’s computation. Formally, we define $Q_{A∪B} = Q_A × Q_B$: each vertex-pair of $M_{A∪B}$ is a 2-tuple, consisting of a vertex of $M_A$ and a vertex of $M_B$ (in that order). The idea is that:

**Assertion 2.2.1.** $M_{A∪B}$ will have destination $q = (q_A, q_B)$ on reading input $w$ exactly if $M_A$ has destination $q_A$ and $M_B$ has destination $q_B$ on reading input $w$.

The definitions of $\text{start}_{A∪B}$, $\delta_{A∪B}$, and $F_{A∪B}$ are now immediate.

- The start vertex-pair of $M$ is the pair consisting of the start vertices of $M_A$ and $M_B$: $\text{start}_{A∪B} = (\text{start}_A, \text{start}_B)$.
- An edge of $M_{A∪B}$, labeled $a$, connects a vertex-pair $p = (p_A, p_B)$ to vertex-pair $q = (q_A, q_B)$, exactly if $p_A$ is connected to $q_A$ by an $a$-labeled edge in $M_A$ and $p_B$ is connected to $q_B$ by an $a$-labeled edge in $M_B$; in other words, $\delta_{A∪B}(p, a) = \delta_{A∪B}((p_A, p_B), a) = (\delta_A(p_A, a), \delta_B(p_B, a)) = (q_A, q_B) = q$.
- The recognizing vertex-pairs of $M_{A∪B}$ are those pairs that include either a recognizing vertex of $M_A$ or a recognizing vertex of $M_B$ (or both): $F = (F_A × Q_B) \cup (Q_A × F_B)$.

**Lemma 2.2.2.** $M_{A∪B}$ recognizes the language $A \cup B$, $L(M_{A∪B}) = A \cup B$ for short.

**Proof.** We need to do two things:

1. Show that if $w$ is recognized by $M_{A∪B}$, i.e. if $w \in L(M_{A∪B})$, then $w \in A \cup B$, and
2. show that if $w \in A \cup B$, then $w \in L(M_{A∪B})$.

The first substep is to verify Assertion 2.2.1. But this is easy to see: consider the computation of $M_{A∪B}$ on input $w$, but only pay attention to what is happening to the first component of its vertex-pair: this is identical to the computation of $M_A$ on input $w$. Likewise, if we pay attention only to the second component of the vertex-pair, we see that the computation is identical to that of $M_B$ on input $w$. In particular, $M_{A∪B}$ has destination vertex-pair $q = (q_A, q_B)$ on reading input $w$ exactly if $M_A$ has destination $q_A$ and $M_B$ has destination $q_B$ after reading input $w$, which is what Assertion 2.2.1 states.

Next, we show (1). So suppose that $M_{A∪B}$ recognizes $w$, that is, on input $w$, $M_{A∪B}$ reaches a recognizing vertex-pair, $q = (q_A, q_B)$ say. By Assertion 2.2.1, $M_{A∪B}$ has destination $q = (q_A, q_B)$ on input $w$ exactly if $M_A$ has destination $q_A$ on input $w$ and $M_B$ has destination $q_B$. Now, by definition, $q$ is a recognizing vertex-pair of $M_{A∪B}$ exactly if $q \in (F_A × Q_B) \cup (Q_A × F_B)$. Consider the case that $q = (q_A, q_B) \in F_A × Q_B$; then $q_A \in F_A$, that is $M_A$ has as destination a recognizing vertex on input $w$, which happens exactly if $w \in A$. Similarly, in the case that $q = (q_A, q_B) \in Q_A × F_B$, $q_B \in F_B$, that is $M_B$ has as destination a recognizing vertex on input $w$, which happens exactly if $w \in B$. This shows that if $M_{A∪B}$ recognizes input $w$, then either $w \in A$ or $w \in B$ (or possibly both); this means that $w \in A \cup B$, which is the statement in (1) above.

Finally, we show (2). If $w \in A \cup B$, then either $w \in A$ or $w \in B$ (or possibly both). Consider the case that $w \in A$. Then $w$ is recognized by $M_A$. That is, on input $w$, $M_A$ has a recognizing vertex as its destination, $q_A$ say, with $q_A \in F_A$. At the same time, $M_B$, on input $w$, has some vertex
$q_B \in Q_B$ as its destination. By Assertion 2.2.1, $M_{A \cup B}$, on input $w$, has as its destination the vertex pair $(q_A, q_B) \in F_A \times Q_B$. So $M_{A \cup B}$ recognizes $w$ since $(q_A, q_B) \in F_A \times Q_B = (F_A \times Q_B) \cup (Q_A \times F_B)$. An essentially identical argument shows that if $w \in B$ then $M_{A \cup B}$ again recognizes $w$. In sum, if $w \in A \cup B$, then $M_{A \cup B}$ recognizes $w$, that is $w \in L(M_{A \cup B})$.

**Corollary 2.2.3.** Let $A$ and $B$ be languages recognized by finite automata. Then there is another finite automata recognizing the language $A \cup B$.

**Recognizing $A \circ B$** Again, we suppose we have finite automata $M_A$ recognizing $A$ and $M_B$ recognizing $B$. We then build a device $M$ to recognize $A \circ B$, or $AB$ for short. $w \in AB$ exactly if we can divide $w$ into two parts, $u$ and $v$, with $u \in A$ and $v \in B$. This immediately suggests how $M$ will try to use $M_A$ and $M_B$. First it runs $M_A$ on substring $u$ and then it runs $M_B$ on substring $v$. The requirements are that $M_A$ recognize $u$ and $M_B$ recognize $v$; that is, on input $u$, $M_A$ goes from its start vertex to a recognizing vertex, and on input $v$, $M_B$ also goes from its start vertex to a recognizing vertex. The difficulty is for $M$ to know when $u$ ends, for there may be multiple possibilities. For example, consider the languages $A = \{\text{strings of one or more } a\text{'s}\}$ and $B = \{\text{strings of one or more } b\text{'s}\}$. On the string $aabb$ it would not be correct to switch from simulating $M_A$ to simulating $M_B$ after reading a single $a$, while it would be correct on the string $abb$. The solution is to keep track of all possibilities. We do not explore this further at this point, as this is more readily understood using the technique of nondeterminism, which is the topic of the next subsection.

Likewise, we defer the description of how to recognize $A^*$ to the next subsection.

### 2.3 Nondeterministic Finite Automata

*Non-deterministic Finite Automata*, NFAs for short, are a generalization of the machines we have already defined, which are often called *Deterministic Finite Automata* by contrast, or DFAs for short. The reason for the name will become clear later.

As with a DFA, an NFA is simply a graph with edges labeled by single letters from the input alphabet $\Sigma$. There is one structural change.

For each vertex $v$ and each character $a \in \Sigma$, the number of edges exiting $v$ labeled with $a$ is unconstrained; it could be 0, 1, 2 or more edges.

This obliges us to redefine what an automaton $M$ is doing, given an input $x$. Quite simply, $M$ on input $x$ determines all the vertices at the end of paths labeled $x$ that begin at the start vertex, *start*. If any of these destination vertices is in the set of Recognizing or Final vertices then $M$ is defined to recognize $x$. Another way of looking at this is that $M$ recognizes $x$ exactly if there is some path (and possibly more than one) labeled $x$, going from *start* to a vertex (state) $q \in F$; such a path is called a *recognizing* or *accepting* path.

**Example 2.3.1.** Let $A = \{w |$ the third to last character in $w$ is a “$b$”$\}$. The machine $M$ with $L(M) = A$ is shown in Figure 2.17.

Note that on input $abbb$ all four vertices are destination vertices, whereas on input $abab$ the second from rightmost vertex is not a destination vertex.
As we will see later, the collection of languages recognized by NFAs is exactly the collection of languages recognized by DFAs. That is, for each NFA $N$ there is a DFA $M$ with $L(M) = L(N)$. Thus NFAs do not provide additional computational power. However, they can be more compact and easier to understand, and as a result they are quite useful.

Indeed, the language from Example 2.3.1 is recognized by the DFA shown in Figure 2.18.

It is helpful to consider how one might implement an NFA $N$. One way is to maintain a bit vector over the vertices, so as to record which vertices are possible current destinations. That is, suppose that on reading input $w$, $N$ can end up at any of the vertices in set $R$, say. The set $S$ of possible destination vertices on reading a further character $a$ is obtained by following all the edges labeled $a$ that exit any of the vertices in $R$. So in the automata of Figure 2.17, the set of possible destination vertices on reading input $b$ is \{“any string”, “last char read is a b”\}, and the set of possible destination vertices on reading $ba$ is \{“any string”, “next to last char read is a b”\}.

Definition 2.3.2 (NFA Recognition). An NFA $M$ recognizes input string $w$ if there is a path from $M$’s start vertex to a recognizing vertex (state) such that the labels along the edges of the path, when concatenated, form the string $w$. That is, were we to follow this path, reading the edge labels as we go, the string read would be exactly $w$; we say this path has label $w$. We call such a path a recognizing or accepting path for $w$. 

Figure 2.17: NFA recognizing \{w \mid\text{the third to last character in } w \text{ is a } b\}.

Figure 2.18: DFA recognizing \{w \mid\text{the third to last character in } w \text{ is a } b\}. By naming the last three characters or all the characters in the string, each vertex specifies the strings for which it is the destination vertex.
The implementation described above keeps track, in turn, of all possible endpoints of paths with labels \( \lambda, w_1, w_1w_2, \ldots, w_1w_2 \ldots w_n \), where \( w = w_1w_2 \ldots w_n \); to recognize \( w \), one needs to have at least one path with label \( w \) leading to a recognizing vertex.

To define NFAs formally, we need to redefine the destination (transition) functions \( \delta \). Now \( \delta \) takes as its arguments a set of vertices (states) \( R \subseteq V \) and a character \( a \in \Sigma \) and produces as output another set of vertices (states) \( S \subseteq V \): \( \delta(R, a) = S \). The meaning is that the set of vertices (states) reachable from \( R \) on reading \( a \) is exactly the set of vertices (states) \( S \).

\( \delta^* \) is also redefined. \( \delta^*(R, w) = S \) means that the set of vertices (states) \( S \) are the possible destinations on reading \( w \) when starting from a vertex (state) in \( R \). So, in particular, \( w \) is recognized by \( M \), \( w \in L(M) \), exactly if \( \delta^*(\{\text{start}\}, w) \cap F \neq \emptyset \); i.e., when starting at vertex \( \text{start} \), on reading \( w \), at least one destination vertex (state) is a recognizing or final vertex (state).

\( \delta \) is often defined formally as follows: \( \delta : 2^Q \times \Sigma \to 2^Q \). In this context, \( 2^Q \) means the collection of all possible subsets of \( Q \), and as \( Q \) is the set of vertices of the NFA, \( \delta \) does exactly what it should. Its inputs are (i) a set of vertices (states), that is one of the elements of the collection \( 2^Q \), and (ii) a character in \( \Sigma \); its output is also a set of vertices (states).

Next, we add one more option to NFAs: edges labeled with the empty string, as shown in Figure 2.19. We call these edges \( \lambda \)-edges for short. The meaning is that if vertex (state) \( p \) is reached, then vertex (state) \( q \) can also be reached without reading any more input. Again, as we shall see later, the same languages as before are recognized by NFAs with \( \lambda \)-edges and by DFAs; however, the \( \lambda \)-edges are very convenient in creating understandable machines. The meanings of the destination functions \( \delta \) and \( \delta^* \) are unchanged.

**Example 2.3.3.** Let \( A \) and \( B \) be languages over the alphabet \( \Sigma \) that are recognized by NFAs \( M_A \) and \( M_B \), respectively. Then the NFA shown in Figure 2.20 recognizes the language \( A \cup B \).

Its first step, prior to reading any input, is to go to the start vertices for machines \( M_A \) and \( M_B \). Then the computations in these two machines are performed simultaneously. The set of recognizing vertices for \( M \) is the union of the recognizing vertices for \( M_A \) and \( M_B \); thus \( M \) reaches a recognizing vertex on input \( w \) exactly if at least one of \( M_A \) or \( M_B \) reaches a recognizing vertex on input \( w \). In other words \( L(M) = L(M_A) \cup L(M_B) \).

**Deterministic vs. Nondeterministic** Another way of viewing the computation of an NFA \( M \) on input \( x \) is that the task is to find a path labeled \( x \) from the start vertex to a recognizing vertex if there is one, and this is done correctly by (inspired) guessing. This process of correct guessing is called *non-deterministic* computation. Correspondingly, if there is no choice or uncertainty in the computation, it is said to be *deterministic*. This is not an implementable view; it is just a convenient way to think about what nondeterminism achieves.
2.3. NONDETERMINISTIC FINITE AUTOMATA

2.3.1 Closure Properties

Definition 2.3.4. A language is said to be regular if it can be recognized by an NFA.

The regular languages obey three closure properties: if $A$ and $B$ are regular, then so are $A \cup B$, $A \circ B$ and $A^*$. We have already seen a demonstration of the first of these. We now give a simpler demonstration of the first property and then show the other two.

First, the following technical lemma is helpful.

Lemma 2.3.5. Let $M$ be an NFA. Then there is another NFA, $N$, which has just one recognizing vertex, and which recognizes the same language as $M$: $L(N) = L(M)$.

Proof. The idea is very simple: $N$ is a copy of $M$ with one new vertex, new_recognize, added; new_recognize is $N$’s only recognizing vertex. $\lambda$-edges are added from the vertices that were recognizing vertices in $M$ to new_recognize, as illustrated in Figure 2.21.

Recall that a recognizing path in an NFA goes from its start vertex to a recognizing vertex. It is now easy to see that $M$ and $N$ recognize the same language. The argument has two parts.

First, any recognizing path in $M$, for string $w$ say, can be extended by a single $\lambda$-edge to reach $N$’s recognizing vertex, and thereby becomes a recognizing path in $N$; in addition, the edge addition leaves the path label unchanged (as $w\lambda = w$). It follows that if $M$ recognizes string $w$ then so does $N$. In other words, $L(M) \subseteq L(N)$. 
Second, removing the last edge from a recognizing path in \( N \) yields a recognizing path in \( M \) having the same path label (for the removed edge has label \( \lambda \)). It follows that if \( N \) recognizes string \( w \) then so does \( M \). In other words, \( L(N) \subseteq L(M) \).

Together, these two parts yield that \( L(M) = L(N) \), as claimed.

We are now ready to demonstrate the closure properties.

**If \( A \) and \( B \) are regular then so is \( A \cup B \)** We show this using the NFA \( M_{A \cup B} \) in Figure 2.22. \( M_{A \cup B} \) is built using \( M_A \) and \( M_B \), NFAs with single recognizing vertices recognizing \( A \) and \( B \), respectively. It has an additional start vertex \( \text{start} \), connected to \( M_A \) and \( M_B \)’s start vertices by \( \lambda \)-edges, and an additional recognizing vertex \( \text{recognize} \), to which \( M_A \) and \( M_B \)’s recognizing vertices are connected by \( \lambda \)-edges.

As in the earlier construction, each recognizing path in \( M_{A \cup B} \) corresponds to a recognizing path in one of \( M_A \) and \( M_B \), and vice-versa, and thus \( w \in L(M_{A \cup B}) \) if and only if \( w \in A \cup B \). We now show this more formally.

**Lemma 2.3.6.** \( M_{A \cup B} \) recognizes \( A \cup B \).

**Proof.** We show that \( A \cup B = L(M_{A \cup B}) \) by showing that \( L(M_{A \cup B}) \subseteq A \cup B \) and that \( A \cup B \subseteq L(M_{A \cup B}) \).

To show that \( L(M_{A \cup B}) \subseteq A \cup B \), it is enough to show that if \( w \in L(M_{A \cup B}) \) then \( w \in A \cup B \) also. So let \( w \in L(M_{A \cup B}) \) and let \( P \) be a recognizing path for \( w \) in \( M_{A \cup B} \). Removing the first and last edges of \( P \), which both have label \( \lambda \), yields a recognizing path in one of \( M_A \) or \( M_B \), also having label \( \lambda \). This shows that if \( w \in L(M_{A \cup B}) \), then either \( w \in L(M_A) = A \) or \( w \in L(M_B) = B \) (of course, it might be in both, but this argument does not reveal this); in other words, if \( w \in L(M_{A \cup B}) \) then \( w \in A \cup B \).

To show that \( A \cup B \subseteq L(M_{A \cup B}) \), it is enough to show that if \( w \in A \cup B \) then \( w \in L(M_{A \cup B}) \) also. Now, if \( w \in A \cup B \), then either \( w \in A \) or \( w \in B \) (or possibly both). Consider the case that \( w \in A \). Then there is a recognizing path \( P_A \) for \( w \) in \( M_A \). Preceding \( P_A \) with the appropriate \( \lambda \)-edge from \( \text{start} \) and following it with the \( \lambda \)-edge to \( \text{recognize} \), yields a recognizing path in \( M_{A \cup B} \), also having label \( w \). This shows that if \( w \in L(M_A) = A \) then \( w \in L(M_{A \cup B}) \) also. Similarly, if \( w \in B \) then \( w \in L(M_{A \cup B}) \) too. Together, this gives that if \( w \in A \cup B \) then \( w \in L(M_{A \cup B}) \).
If $A$ and $B$ are regular then so is $A \circ B$. We show this using the NFA $M_{AB}$ displayed in Figure 2.23. $M_{AB}$ is built using $M_A$ and $M_B$, NFAs with single recognizing vertices, recognizing $A$ and $B$,

![Diagram of $M_{AB}$](image)

Figure 2.23: NFA $M_{AB}$ recognizing $A \circ B$

respectively. $M_{AB}$ comprises a copy of $M_A$ plus a copy $M_B$, plus one additional edge. $M_A$’s start vertex is also $M_{AB}$’s start vertex, and $M_B$’s recognizing vertex is $M_{AB}$’s only recognizing vertex. Finally, the new edge, $e$, joins $M_A$’s recognizing vertex to $M_B$’s start vertex.

The idea of the construction is that a recognizing path in $M_{AB}$ corresponds to recognizing paths in $M_A$ and $M_B$ joined by edge $e$. It then follows that $w \in L(M_{AB})$ if and only if $w$ is the concatenation of strings $u$ and $v$, $w = uv$, with $u \in A$ and $v \in B$. We now show this more formally.

**Lemma 2.3.7.** $M_{AB}$ recognizes $A \circ B$.

**Proof.** We show that $L(M_{AB}) = A \circ B$ by showing that $L(M_{AB}) \subseteq A \circ B$ and that $A \circ B \subseteq L(M_{AB})$.

To show that $L(M_{AB}) \subseteq A \circ B$, it is enough to show that if $w \in L(M_{AB})$ then $w \in A \circ B$ also. So let $w \in L(M_{AB})$ and let $P$ be a recognizing path for $w$ in $M_{AB}$. Removing edge $e$ from $P$ creates two paths $P_A$ and $P_B$, with $P_A$ being a recognizing path in $M_A$ and $P_B$ a recognizing path in $M_B$. Let $u$ and $v$ be the path labels for $P_A$ and $P_B$, respectively. So $u \in A$ and $v \in B$. As $e$ is a $\lambda$-edge, $w = u\lambda v = uv$. This shows that $w \in A \circ B$.

To show that $A \circ B \subseteq L(M_{AB})$, it is enough to show that if $u \in A$ and $v \in B$ then $uv \in L(M_{AB})$. So let $u \in A$, $v \in B$, let $P_A$ be a recognizing path for $u$ in $M_A$, and let $P_B$ be a recognizing path for $v$ in $M_B$. Then form the path $P = P_A, e, P_B$ in $M_{AB}$; clearly, this is a recognizing path. Further, it has label $u\lambda v = uv$. So $uv \in L(M_{AB})$. □

If $A$ is regular then so is $A^*$. We show this using the NFA $M_{A^*}$ in Figure 2.24.

$M_{A^*}$ is built using $M_A$, an NFA with a single recognizing vertex recognizing $A$. $M_{A^*}$ comprises a copy of $M_A$ plus a new start vertex, plus two additional $\lambda$-edges, $e$ and $f$. $e$ joins $M_A$’s start vertex to $M_A$’s start vertex, and $f$ joins $M_A$’s recognizing vertex to $M_A$’s start vertex. $M_A$’s start vertex is also its recognizing vertex.

The construction is based on the observation that removing all copies of $e$ and $f$ from a recognizing path in $M_{A^*}$ yields subpaths, $k$ of them say, each of which is a recognizing path in $M_A$. It then follows that a string $w$ is recognized by $M_{A^*}$ if and only if it is the concatenation of $k$ strings recognized by $M_A$, for some $k \geq 0$. Now we show this more formally.

**Lemma 2.3.8.** $M_{A^*}$ recognizes $A^*$.

**Proof.** We show that $L(M_{A^*}) = A^*$ by showing that $L(M_{A^*}) \subseteq A^*$ and that $A^* \subseteq L(M_{A^*})$. 
To show that \( L(M_{A^*}) \subseteq A^* \), it is enough to show that if \( w \in L(M_{A^*}) \) then \( w \in A^* \) also. So let \( w \in L(M_{A^*}) \) and let \( P \) be a recognizing path for \( w \) in \( M_{A^*} \). Removing all instances of edges \( e \) and \( f \) from \( P \) creates \( k \) subpaths, for some \( k \geq 0 \), where each subpath is a recognizing path in \( M_A \). Let the path labels on these \( k \) subpaths be \( u_1, \ldots, u_k \), respectively. So \( u_i \in A \), for \( 1 \leq i \leq k \). As \( e \) and \( f \) are \( \lambda \)-edges, \( w = \lambda u_1 \lambda u_2 \lambda \cdots \lambda u_k \lambda = u_1 u_2 \cdots u_k \). Thus \( w \in A^* \).

To show that \( A^* \subseteq L(M_{A^*}) \), it is enough to show that if \( w \in A^* \) then \( w \in L(M_{A^*}) \). If \( w \in A^* \) then \( w = u_1 u_2 \cdots u_k \), with \( u_1, u_2, \ldots, u_k \in A \), for some \( k \geq 0 \). As \( u_i \in A \), there is a recognizing path \( P_i \) in \( M_A \) for \( u_i \). Let \( P \) be the following path in \( M_{A^*} \): \( e, P_1, e, P_2, \ldots, e, P_k, f \) (for \( k = 0 \) we intend the path of zero edges). \( P \) is a recognizing path in \( M_{A^*} \), and it has label \( u_1 u_2 \cdots u_k = w \), as \( e \) and \( f \) are \( \lambda \)-edges. Thus \( w \in L(M_{A^*}) \).

\[
\]

### 2.3.2 Every regular language is recognized by a DFA

Let \( N \) be an NFA. We show how to construct a DFA \( M \) with \( L(M) = L(N) \).

Recall that to implement the computation of an NFA \( N \) on an input \( w \) we keep track of the set of current destination vertices as \( w \) is read. Suppose that \( S_0, S_1, \ldots, S_n \) is the sequence of sets of destination vertices reached on reading \( w = w_1 w_2 \cdots w_n \) character by character; i.e., \( S_i \) is the set of destination vertices for input \( w_1 w_2 \cdots w_i \), and in particular \( S_0 \) is the set of vertices reachable without reading any input (i.e. on input \( \lambda \)). \( M \) will need to remember exactly this information: the set of current destination vertices in \( N \). To do this, we make the vertices in \( M \) be the possible sets of destination vertices in \( N \). To avoid confusion, henceforth we call the vertices in \( M \) supervertices. If \( N \) has \( q \) vertices, \( M \) has \( 2^q \) supervertices, one for each subset of \( N \). In other words \( M \)'s collection (set) of supervertices is the power set of \( Q \), \( 2^Q \).

The relation between the supervertices in \( M \) and the sets of vertices in \( N \) is specified by the following assertion.

**Assertion 2.3.9.** On input \( w \), \( N \) goes through the sequence of destination sets \( S_0, S_1, S_2, \ldots, S_n \) if and only if \( M \) traverses the path \( S_0, S_1, S_2, \ldots, S_n \).

To achieve Assertion 2.3.9, we define the transition function \( \delta \) for \( M \) to be identical to the transition function \( \delta \) for \( N \). Note that this specifies where \( M \)'s edges go. More formally, if \( U = \{ u_1, u_2, \ldots, u_k \} \) is a supervertex of \( M \), and in \( N \) \( \delta(u_i, a) = W_i \), for \( 1 \leq i \leq k \), then in \( M \), \( \delta(U, a) = \cup_{1 \leq i \leq k} W_i \).
To make sure the assertion is correct initially, that is for \( w = \lambda \), we set \( M \)'s start supervertex to be the set of \( N \)'s vertices that \( N \) can reach on input \( \lambda \).

Finally, to ensure that \( L(M) = L(N) \), we recall that \( N \) accepts input \( w \) if its destination set \( S \) on input \( w \) includes a vertex in \( F \), the set of its recognizing vertices; accordingly, we define the recognizing supervertices of \( M \) to be those supervertices that include one or more of the recognizing vertices of \( N \); i.e. \( R \) is a recognizing supervertex for \( M \) exactly if \( R \cap F \neq \phi \).

We illustrate the construction in Figure 2.25.

![Figure 2.25: NFA and corresponding DFA. The strings that terminate at a vertex of the DFA, e.g. at the vertex labeled \( \{p, q, s\} \), are exactly those strings that can terminate at all of the corresponding nodes in the NFA, the vertices labeled \( p \), \( q \), and \( s \) in this example.](image)

**Lemma 2.3.10.** \( L(M) = L(N) \).

*Proof.** Clearly Assertion 2.3.9 is true, for \( N \) starts, prior to any reads, with destination set \( S_0 \), and \( M \) starts at supervertex \( S_0 \). Then, as the machines use the same transition function \( \delta \), the destination set that \( N \) reaches after reading each successive input character will be the destination supervertex for \( M \) after reading the same input.

It remains to consider which strings the two machines recognize. \( M \) recognizes input \( w \) if and only if on input \( w \) it reaches supervertex \( R \) where \( R \cap F \neq \phi \); and \( N \) recognizes input \( w \) if and only if its set \( S \) of possible destinations satisfies \( S \cap F \neq \phi \). But we have shown that \( S = R \). So the two machines recognize the same collection of strings, that is \( L(M) = L(N) \). \( \square \)

**Implementation Remark** The advantage of an NFA is that it may have far fewer vertices (states) than a DFA recognizing the same language, and thus use much less memory to store them. On the other hand, when running the machines, the NFA may be less efficient, as the set of reachable vertices had to be computed as the input is read, whereas in the DFA one just needs
to follow a single edge. Which choice is better depends on the particulars of the language and the implementation environment.

2.4 Non Regular Languages

We turn now to a method for demonstrating that some languages are not regular. For example, as we shall see, \( L = \{a^n b^n | n \geq 1 \} \) is not a regular language. Intuition suggests that to recognize \( L \) we would need to count the number of \( a \)'s in the input string; in turn, this suggests that any automata recognizing \( L \) would need an unbounded number of vertices. But how do we turn this into a convincing argument?

We use a proof by contradiction. Suppose for a contradiction that \( L \) were regular. Then there must be a DFA \( M \) that accepts \( L \). \( M \) has some number \( k \) of vertices. Let's consider feeding \( M \) the input \( a^k b^k \). Look at the sequence of \( k+1 \) vertices \( M \) goes through on reading \( a^k \): \( r_0 = \text{start}, r_1, r_2, \ldots, r_k \), where some of the \( r_i \)'s may be a repeated instance of the same vertex. This is illustrated in Figure 2.26.

As there are only \( k \) distinct vertices, there is at least one vertex that is visited twice in this sequence, \( r_h = r_j \), say, for some pair \( h, j \), \( 0 \leq h < j \leq k \). This is shown in Figure 2.27.

In fact we see that \( r_{j+1} = r_{h+1} \) if \( j + 1 \leq k \), \( r_{j+2} = r_{h+2} \) if \( j + 2 \leq k \), and so forth. But all we will need for our result is the presence of one loop in the path, so we will stick with the representation in Figure 2.27.

It is helpful to partition the input into four pieces: \( a^i, a^{j-i}, a^{k-j}, b^k \). The first \( a^i \) takes \( M \) from vertex \( r_0 \) to \( r_i \) (to the start of the loop), the next \( a^{j-i} \) takes \( M \) from \( r_i \) to \( r_j \) (once around the loop), the final \( a^{k-j} \) takes \( M \) from \( r_j \) to \( r_k \), and \( b^k \) takes \( M \) from \( r_k \) to a recognizing vertex, as shown in figure 2.28.

What happens on input \( a^i a^{j-i} a^{j-i} a^{k-j} b^k = a^{k+j-i} b^k \)? The initial \( a^i \) takes \( M \) from \( r_0 \) to \( r_i \), the first \( a^{j-i} \) takes \( M \) from \( r_i \) to \( r_j = r_i \) (once around the loop), the second \( a^{j-i} \) takes \( M \) from \( r_i \) to \( r_j \) (around the loop again), the \( a^{k-j} \) takes \( M \) from \( r_j \) to \( r_k \), and the \( b^k \) takes \( M \) from \( r_k \) to a recognizing vertex. So \( M \) accepts \( a^{k+j-i} b^k \), which is not in \( L \) as \( j - i > 0 \). This is a contradiction, and thus the initial assumption, that \( L \) was regular, must be incorrect.

We now formalize the above approach in the following lemma.
2.4. NON REGULAR LANGUAGES

![Diagram](image)

Figure 2.28: The path traversed on input $a^k$.

**Lemma 2.4.1.** (Pumping Lemma) Let $L$ be a regular language. Then there is an integer $p = p_L \geq 1$, the pumping length for $L$, with the property that for each string $s$ in $L$ of length at least $p$, $s$ can be pumped, that is $s$ can be written as the concatenation of 3 substrings $x$, $y$, and $z$, that is $s = xyz$, and these substrings satisfy the following conditions.

1. $|y| > 0$,
2. $|xy| \leq p$,
3. For each integer $i \geq 0$, $xy^iz \in L$.

**Proof.** Again, we use a proof by contradiction. So suppose that $L$ were regular and let $M$ be a DFA accepting $L$. Now suppose that $M$ has $p$ vertices (states), $p$, the number of vertices in $M$, will be the pumping length for $L$.

Let $s$ be a string in $L$ of length $n \geq p$. Write $s$ as $s = s_1s_2\cdots s_n$, where each $s_h$, $1 \leq h \leq n$, is a character in $\Sigma$, the alphabet for $L$. Consider the substring $s' = s_1s_2\cdots s_p$. We look at the path $M$ follows on input $s'$. It must go through $p + 1$ vertices, and as $M$ has only $p$ vertices, at least one vertex must be repeated. Let start = $r_0$, $r_1$, $\cdots$, $r_p$ be this sequence of vertices and suppose that $r_h = r_j$, where $0 \leq h < j \leq p$, is a repeated vertex, as shown in Figure 2.29.

![Diagram](image)

Figure 2.29: The path traversed on input $s' = s_1s_2\cdots s_p$.

Let $x$ denote $s_1\cdots s_h$, $y$ denote $s_{h+1}\cdots s_j$, and $z$ denote $s_{j+1}\cdots s_p s_{p+1}\cdots s_n$. The path traversed on input $s = xyz$ is illustrated in Figure 2.30.

As $j > i$, $|y| = j - i > 0$. Also $|xy| = |s_1\cdots s_j| = j \leq p$. So (1) and (2) are true.

Clearly, $xz$ is recognized by $M$, for $x$ takes $M$ from $r_0$ to $r_h = r_j$ and $z$ takes $M$ from $r_j$ to a recognizing vertex.

Similarly $xy'z$ is recognized by $M$ for any $i \geq 1$, for $x$ takes $M$ from $r_0$ to $r_h$, each repetition of $y$ takes $M$ from $r_h$ (back) to $r_j = r_h$, and then the $z$ takes $M$ from $r_j$ to a recognizing vertex. Thus (3) is also true, proving the result. \qed
Now, to show that a language \( L \) is non-regular we use the pumping Lemma in the following way. We begin by assuming that \( L \) is regular so as to obtain a contradiction. Next, we assert that there is a pumping length \( p \) such that for each string \( s \) in \( L \) of length at least \( p \) the three conditions of the Pumping Lemma hold. The next (and more substantial) task is to choose a particular string \( s \) to which we will apply the conditions of the Pumping Lemma, and condition (3) in particular, so as to obtain a contradiction.

Example 2.4.2. Let us look at the language \( L = \{a^n b^n \mid n \geq 1\} \) again. The argument showing that \( L \) is not regular proceeds as follows.

**Step 1.** Suppose, so as to obtain a contradiction, that \( L \) were regular. Then \( L \) must have a pumping length \( p > 1 \) such that for any string \( s \in L \) with \( |s| \geq p \), \( s \) can be pumped.

**Step 2.** Choose \( s \). Recall that we chose the string \( s = a^p b^p \).

**Step 3.** By pumping \( s \), obtain a contradiction.

As \( s \) can be pumped (for \( s \in L \) and \( |s| = 2p \geq p \)), we can write \( s = xyz \), with \( |y| > 0 \), \( |xy| \leq p \), and \( xy^i z \in L \) for every integer \( i \geq 0 \).

As the first \( p \) characters of \( s \) are all \( a \)'s, the substring \( xy \) must also be a string of all \( a \)'s. By condition (3), with \( i = 0 \), we have that \( xz \in L \); but the string \( xz \) has removed \( |y| \) \( a \)'s from \( s \), that is \( xz = a^{p-|y|} b^p \), and this string is not in \( L \) since \( p - |y| \neq p \). This is a contradiction for we have shown that \( xz \in L \) and \( xz \notin L \).

**Step 4.** Consequently the initial assumption is incorrect; that is, \( L \) cannot be recognized by a DFA, so \( L \) is not regular.

Let me stress the sequence in which the argument goes. First the existence of pumping length \( p \) for \( L \) is asserted (different regular languages \( L \) may have different pumping lengths, but each regular language containing arbitrarily long strings will have a pumping length). Then a suitable string \( s \) is chosen. The length of \( s \) will be a function of \( p \). \( s \) is chosen so that when it is pumped a string outside of \( L \) is obtained. An important point about the pumped substring \( y \) is that while we know \( y \) occurs among the first \( p \) characters of \( s \), we do not know exactly which ones form the substring \( y \). Consequently, a contradiction must arise for every possible substring \( y \) of the first \( p \) characters in order to show that \( L \) is not regular.

There is a distinction here: for a given value of \( p \), \( s \) is fully determined; for instance, in Example 2.4.2 for \( p = 3 \), \( s = aaabbb \). By contrast, all that we know about \( x \) and \( y \) is that together they contain between 1 and 3 characters, and that \( y \) has at least one character. There are 6 possibilities in all for the pair \((x, y)\), namely: \((\lambda, a)\), \((\lambda, aa)\), \((\lambda, a\lambda)\), \((a, a)\), \((a, aa)\), \((aa, a)\). In general, for \( |xy| \leq p \), there are \( \frac{1}{2} p(p + 1) \) choices of \( x \) and \( y \). The argument leading to a contradiction must
work for every possible choice of \( p \) and every possible partition of \( s \) into \( x, y, \) and \( z \). Of course, you are not going to give a separate argument for each case given that there are infinitely many cases. Rather, the argument must work regardless of the value of \( p \) and regardless of which partition of \( s \) is being considered.

Next, we show an alternative Step 3 for Example 2.4.2.

**Alternative Step 3.** In applying Condition 3, use \( i = 2 \) (instead of \( i = 0 \)), giving \( xyyz \in L \). But \( xyyz \) adds \( |y| \) a’s to \( xyz \), namely \( xyyz = a^{p+|y|}b^p \), and this string is not in \( L \) since \( p + |y| \neq p \). This is a contradiction for we have shown that \( xyyz \in L \) and \( xyyz \notin L \).

In Example 2.4.2 both pumping down \( (i = 0) \) and pumping up \( (i = 2) \) will yield a contradiction. This is not the case in every example. Sometimes only one direction works, and then only for the right choice of \( s \).

**A common mistake.** Not infrequently, an attempted solution may try to specify how \( s \) is partitioned into \( x, y, \) and \( z \). In Example 2.4.2, this might take the form of stating that \( x = \lambda, y = a^p, \) and \( z = b^p \), and then obtaining a contradiction for this partition. This is an incomplete argument, however. All that the Pumping Lemma states is that there is a partition; it does not tell you what the partition is. *The argument showing a contradiction must work for every possible partition.*

---

**Example 2.4.3.** Let \( K = \{ww^R | w \in \{a, b\}^* \}. \) For example, \( \lambda, abba, abababa \in K, \) \( ab, a, aba \notin K \). We show that \( K \) is not regular.

**Step 1.** Suppose, so as to obtain a contradiction, that \( K \) were regular. Then \( K \) must have a pumping length \( p \geq 1 \) such that for any string \( s \in K \) with \( |s| \geq p \), \( s \) can be pumped.

**Step 2.** Choose \( s \). We choose the string \( s = a^pba^p \).

**Step 3.** By pumping \( s \), obtain a contradiction.

As \( s \) can be pumped (for \( |s| = 2p + 2 \geq p \)), we can write \( s = xyz \), with \( |y| > 0, |xy| \leq p, \) and \( xy^iz \in K \) for every integer \( i \geq 0 \).

As the first \( p \) characters of \( s \) are all \( a \)’s, the substring \( xy \) must also be a string of all \( a \)’s. By condition (3), with \( i = 0 \), we have that \( xx \in K \); but the string \( xx \) has removed \( |y| \) a’s from \( s \), that is \( xx = a^{p-|y|}bba^p \), and this string is not in \( K \) since it is not in the form \( ww^R \) for any \( w \).

(To see this, note that if it were in the form \( ww^R \), as \( xx \) contains two \( b \)’s, one of them would be in the \( w \) and the other in the \( w^R \); this forces \( w = a^{p-|y|}b \) and \( w^R = ba^p \), and this is not possible as \( p - |y| \neq p \). This is a very detailed explanation, which we will not spell out to this extent in future. Noting that \( p - |y| \neq p \) will suffice.)

This is a contradiction for we have shown that \( xx \in K \) and \( xx \notin K \).

**Step 4.** Consequently the initial assumption is incorrect; that is, \( K \) cannot be recognized by a DFA, so \( K \) is not regular.

**Another common mistake.** I have seen attempted solutions for the above example, Example 2.4.3, that set \( s = w^p(w^R)^p \). This is not a legitimate definition of \( s \). For \( w \) is an arbitrary string, so such a definition does not fully specify the string \( s \), that is it does not spell out the characters forming \( s \). To effectively apply the Pumping Lemma you are going to need to choose an \( s \) in which only \( p \) is left unspecified. So if you are told, for example, that \( p = 3 \), then you must be able to write down \( s \) as a specific string of characters (in Example 2.4.3, with \( p = 3, s = aaabbaa \)).

The assumption in the application of the Pumping Lemma is that the language \( L \) under consideration is recognized by a DFA. What is not known is the assumed size of the DFA. What the
argument leading to a contradiction shows is that regardless of its size, the supposed DFA cannot recognize \( L \), but this requires the argument to work for any value of \( p \).

**Example 2.4.4.** \( H = \{a^{i^2} \mid i \geq 0 \} \). We show that \( K \) is not regular. To do this we introduce another new technique. Note that the gap between successive strings in \( H \) is growing (when we order these strings by increasing length). In particular, \(|a^{(i+1)^2}| - |a^{i^2}| = (i + 1)^2 - i^2 = 2i + 1\). We will use this to show that when pumping up, for large enough \( i \), we will produce a string \( s' \notin H \). We proceed as follows.

**Step 1.** Suppose, so as to obtain a contradiction, that \( H \) were regular. Then \( H \) must have a pumping length \( p \geq 1 \) such that for any string \( s \in K \) with \(|s| \geq p \), \( s \) can be pumped.

**Step 2.** Choose \( s \). We choose the string \( s = a^{p^2} \).

**Step 3.** By pumping \( s \), obtain a contradiction.

As \( s \) can be pumped (for \(|s| = p^2 \geq p\)), we can write \( s = xyz \), with \(|y| > 0\), \(|xy| \leq p\), and \( xy^iz \in H \) for every integer \( i \geq 0 \).

By condition (3), with \( i = 2 \), we have that \( s' = xyyz \in H \). As \( 1\lambda|y| \leq p \), \(|s| < |s'| \leq |s| + p = p^2 + p < (p + 1)^2 \). Thus \(|s'| \) lies strictly between \(|s| \) and the length of the next string in \( H \) in increasing length order. But this means \( s' \notin H \). This is a contradiction for we have shown that \( s' \in H \) and \( s' \notin H \).

**Step 4.** Consequently the initial assumption is incorrect; that is, \( H \) cannot be recognized by a DFA, so \( H \) is not regular.

**Example 2.4.5.** \( J = \{w \mid w \) has equal numbers of \( a \)'s and \( b \)'s\}. We show that \( J \) is not regular. To do this we introduce a new technique. Note that if \( A \) and \( B \) are regular then so is \( A \cap B = (\overline{A} \cup \overline{B}) \) (for the union and complement of regular languages are themselves regular; alternatively, see Problem 3).

Now note that \( R = \{a^ib^j \mid i, j \geq 0\} \) is regular. Thus if \( J \) were regular, then \( J \cap R = \{a^ib^j \mid i \geq 0\} \) would also be regular. But we have already shown that \( J \cap R \) is not regular in Example 2.4.2. Thus \( J \) cannot be regular either. (Strictly, this is a proof by contradiction.)

We could also proceed as in Example 2.4.2, applying the Pumping Lemma to string \( s = a^pb^p \). The exact same argument will work.

**Question** Does the proof of the Pumping Lemma work if we consider a \( p \)-state NFA that accepts \( L \), rather than a \( p \)-state DFA? Justify your answer.

### 2.5 Regular Expressions

Regular expressions provide another, elegant and simple way of describing regular languages. The reader may wish to review Section 2.1 at this point.

Next we show that regular expressions represent exactly the regular languages.

**Lemma 2.5.1.** Let \( r \) be a regular expression. There is an NFA \( N_r \) that recognizes the language represented by \( r \): \( L(N_r) = L(r) \).
2.5. REGULAR EXPRESSIONS

Proof. The proof is by induction on the number of operators (union, concatenation, and star) in regular expression \( r \).

The base case is for zero operators, which are also the base cases for specifying regular expressions. It is easy to give DFAs that recognize the languages specified in each of these base cases and this is left as an exercise for the reader.

For the inductive step, suppose that \( r \) is given by one of the recursive definitions, \( r_1 \cup r_2 \), \( r_1 \circ r_2 \), or \( r_1^* \). Since \( r_1 \) and \( r_2 \) if it occurs, contain fewer operators than \( r \), we can assume by the inductive hypothesis that there are NFAs recognizing the languages represented by regular expressions \( r_1 \) and \( r_2 \). Then Lemmas 2.3.6–2.3.8 provide the NFAs recognizing the languages \( r \).

We can conclude that there is an NFA recognizing \( L(r) \).

To prove the converse, that every regular language can be represented by a regular expression takes more effort. To this end, we introduce yet another variant of NFAs, called GNFA (for Generalized NFAs).

In a GNFA each edge is labeled by a regular expression \( r \) rather than by one of \( \lambda \) or a character \( a \in \Sigma \). We can think of an edge labeled by regular expression \( r \) being traversable on reading string \( x \) exactly if \( x \) is in the language represented by \( r \). String \( w \) is recognized by a GNFA \( M \) if there is a path \( P \) in \( M \) from its start vertex to a recognizing vertex such that \( P \)'s label, the concatenation of the labels on \( P \)'s edges, forms a regular expression that includes \( w \) among the set of strings it represents.

In more detail, suppose \( P \) consists of edges \( e_1, e_2, \ldots, e_k \), with labels \( r_1, r_2, \ldots, r_k \), respectively; then \( P \) is a \( w \)-recognizing path if \( w \) can be written as \( w = w_1 w_2 \cdots w_k \) and each \( w_i \) is in the language represented by \( r_i \), for \( 1 \leq i \leq k \).

Clearly, every NFA is a GNFA, so it will be enough to show that any language recognized by a GNFA can also be represented by a regular expression.

We begin with two simple technical lemmas.

Lemma 2.5.2. Suppose that GNFA \( M \) has two vertices \( u \) and \( v \) with \( h \) edges from \( u \) to \( v \), edges \( e_1, e_2, \ldots, e_h \). Suppose further that these edges are labeled by regular expressions \( r_1, r_2, \ldots, r_h \), respectively. Then replacing these \( h \) edges by a new edge \( e \) labeled \( r_1 \cup r_2 \cup \cdots \cup r_h \), or \( r \) for short, yields a GNFA \( N \) recognizing the same language as \( M \).

\[
\begin{array}{c}
\text{Figure 2.31: Edge replacement in Lemma 2.5.2.}
\end{array}
\]

Proof. The edge replacement is illustrated in Figure 2.31.

First we show that \( L(M) \subseteq L(N) \). Suppose that \( M \) recognizes string \( w \). Let \( P \) be a \( w \)-recognizing path in \( M \). Replacing each instance of edge \( e_i \), for \( 1 \leq i \leq h \), by edge \( e \) yields a new path \( P' \) in \( N \), with \( P' \) being \( w \)-recognizing (for a substring \( w_i \) read on traversing edge \( e_i \), and hence represented by \( r_i \), is also represented by \( r \) and so can be read on traversing edge \( e \) also. This shows \( w \in L(N) \).
Next we show that \(L(N) \subseteq L(M)\). Suppose that \(N\) recognizes string \(w\). Let \(P\) be a \(w\)-recognizing path in \(N\). Note that as \(q\) is a non-recognizing vertex in \(M\), \(q\) cannot be the last vertex on \(P\). Suppose that edge \(e\) is traversed \(j\) times in \(P\), and let \(w_1, w_2, \ldots, w_j\) be the substrings of \(w\) read on these \(j\) traversals. Each substring \(w_g, 1 \leq g \leq j\), is represented by regular expression \(r\), and as \(r = r_1 \cup \cdots \cup r_h\), each \(w_g\) is represented by one of \(r_1, r_2, \ldots, r_h\); so let \(w_g\) be represented by \(r_{i_g}\), where \(1 \leq i_g \leq h\). Replacing the corresponding instance of \(e\) in \(P\) by \(e_{i_g}\) yields a path \(P'\) in \(M\) which is also \(w\)-recognizing. This shows that \(w \in L(M)\).

**Lemma 2.5.3.** Suppose that GNFA \(M\) has three vertices \(u, v, q\), connected by edges labeled as shown in Figure 2.32 and further suppose that they are the only edges incident on \(q\). Finally, suppose that \(q\) is not a recognizing vertex.

![Figure 2.32: Edge replacement in Lemma 2.5.3.](image)

Then removing vertex \(q\) and creating edge \((u, v)\) with label \(r_1r_2^*r_3\) yields a GNFA \(N\) recognizing the same language as \(M\).

**Proof.** First we show that \(L(M) \subseteq L(N)\). Suppose that \(w \in L(M)\), and let \(P\) be a \(w\)-recognizing path in \(M\). We build a \(w\)-recognizing path \(P'\) in \(N\). Consider a segment of the path from \(u\) to \(v\) going through \(q\) in \(P\). It consists of edge \((u, q)\), followed by some \(k \geq 0\) repetitions of edge \((q, q)\), followed by edge \((q, v)\). This subpath has label \(r_1r_2^kr_3\). All strings represented by \(r_1r_2^kr_3\) are represented by \(r_1r_2^*r_3\), and so we can replace this subpath by the new edge \((u, v)\) in \(N\). Thus \(w\) is recognized by \(N\).

Next, we show that \(L(N) \subseteq L(M)\). Suppose that \(w \in L(N)\), and let \(P\) be a \(w\)-recognizing path in \(N\). We build a \(w\)-recognizing path \(P''\) in \(M\). Consider an instance of edge \(e\) with \(e = (u, v)\) on \(P\), if any. Suppose string \(w_l\) is read on traversing \(e\). Then \(w_l\) is in the language represented by \(r_1r_2^kr_3\) for some \(k \geq 0\) (for this is what \(r_1r_2^*r_3\) means: \(r_1r_2^*r_3 = r_1r_3 \cup r_1r_2r_3 \cup r_1r_2^2r_3 \cup r_1r_2^3r_3 \cup \cdots\)). So we can replace this instance of edge \(e\) with the edge sequence \((u, q)\) followed by \(k\) instances of edge \((q, q)\), followed by edge \((q, v)\). When all instances of \(e\) in \(P\) are replaced in this manner, the resulting path \(P''\) in \(M\) is still \(w\)-recognizing, so \(w \in M\). \(\square\)

**Lemma 2.5.4.** Let \(M\) be a GNFA. Then there is a regular expression \(r\) representing the language recognized by \(M\): \(L(r) = L(M)\).

**Proof.** We begin by modifying \(M\) so that its start vertex has no in-edge (if need be by introducing a new start vertex) and so that it has a single recognizing vertex with no out-edges (again, by adding a new vertex, if needed). Also, for each pair of vertices, if there are several edges between them, they are replaced by a single edge by applying Lemma 2.5.2. Let \(N_0\) be the resulting machine and suppose that it has \(n + 2\) vertices \(\{\text{start}, q_1, q_2, \ldots, q_n, \text{recognize}\}\).

We construct a sequence \(N_1, N_2, \ldots, N_n\) of GNFAs, where \(N_i\) is \(N_{i-1}\) with \(q_i\) removed and otherwise modified so that \(N_{i-1}\) and \(N_i\) recognize the same language, for \(1 \leq i \leq n\).
2.5. REGULAR EXPRESSIONS

Thus \( L(N_n) = L(M) \). But \( L_n \) has two vertices, \textit{start} and \textit{recognize}, joined by a single edge labeled by a regular expression \( r \), say. Clearly, the language recognized by \( N_n \) is \( L(r) \). So it remains to show how to construct \( N_i \) given \( N_{i-1} \).

\textbf{Step 1.} Let \( q = q_i \) be the vertex being removed. For each pair of vertices \( u, v \neq q \) such that there is a path \( u, q, v \) we make a new copy \( q_{u, v} \) of \( q \) as shown in Figure 2.33, and then remove the vertex \( q \) and the edges incident on it.

![Figure 2.33: Vertex duplication in Step 1.](image)

Clearly, a path segment \( (u, q \text{ repeated } k \geq 1 \text{ times}, v) \) in \( N_{i-1} \) has the same label as the path segment \( (u, q_{u, v} \text{ repeated } k \text{ times}, v) \), and consequently the labels on the recognizing paths in \( N_{i-1} \) and the machine following the Step 1 modification are the same.

\textbf{Step 2.} In turn, for each vertex \( q_{u, v} \), apply Lemmas 2.5.3 and 2.5.2 to the subgraph formed by \( u, q_{u, v}, \) and \( v \), for each pair of vertices \( u, v \). The resulting GNFA is \( N_i \). Clearly, \( L(N_i) = L(N_{i-1}) \). The effect of the application of these lemmas is illustrated in Figure 2.34. Note that in fact we could have performed the change achieved by Step 2 without introducing the vertices \( q_{u, v} \). They are here just to simplify the explanation.

![Figure 2.34: Change due to Step 2.](image)

\[ \]

2.5.1 Finding a Pattern in a Text

We are now ready to return to the problem described at the start of the chapter. Let \( r \) be a regular expression and let \( t \), the text, be a string over the same alphabet. How do we identify every
occurrence of strings \( s \in L(r) \) in the text \( t \)? In other words, we want to identify every substring \( s \) of \( t \) such that \( s \in L(r) \). A typical way of identifying a substring is to give the indices in \( t \) of its first and last characters.

We have just shown that for every regular \( r \), there is an NFA \( N(r) \) such that \( L(r) = L(N(r)) \), so to test if a string is in \( L(r) \) it suffices to test if it is recognized by \( N(r) \). In particular, we could test whether \( t \in L(N(r)) \) in \( O(|t|) \) time by giving \( t \) as input to \( N(r) \). However, the question we want to solve is to identify all substrings of \( t \) that are in \( L(N(r)) \).

Suppose we modify the finite automata so that they can provide output whenever a vertex is reached. In particular, suppose that whenever a recognizing vertex is reached, the automata could output \( R \) (for recognize) and suppose we are independently tracking the index of the last character read. Then we would be able to identify every substring starting at the first character in \( t \) that lies in \( L(N(r)) = L(r) \), and further this could still be done in \( O(|t|) \) time.

However, in general, we are not going to be able to report every substring in \( L(r) \) that lies in \( t \) in time \( O(|t|) \). For suppose \( r = a^* \), strings of one or more \( a \)'s. If \( t = a^n \), then \( t \) has \( \frac{1}{2}n(n-1) \) substrings in \( L(r) \) so reporting them (unless one finds a compact way of specifying them) will take \( \Theta(n^2) \) time.

One solution is to run the just described algorithm \( |t| \) times, starting, in turn, at the first character in \( t \), at the second character, \ldots, at the \( |t| \)th character. One small but practical optimization is to stop each iteration once the only reachable vertices are sink vertices.

### 2.6 Correctness of Vertex Specifications

Given a Finite Automata, with a vertex specification \( S_v \) for each vertex \( v \), how can one determine whether the vertex specifications are correct, in the sense of exactly specifying those strings for which the automata can reach \( v \), when starting at the start vertex?

**Definition 2.6.1.** A specification \( S_v \) for a vertex \( v \) in a DFA or NFA is a set of strings \( s \).

- e.g. \( S_v = \{ \text{even length strings} \} \).

  For the specifications to be correct they will need to satisfy three properties, including forward and backward consistency, which we define next. We begin by considering automata with no \( \lambda \)-edges.

  Let \( e = (u, v) \) be an edge of the finite automata labeled by \( a \). Suppose that \( u \) and \( v \) have specifications \( S_u \) and \( S_v \), respectively.

**Definition 2.6.2.** \( S_u \) and \( S_v \) are forward-consistent with respect to edge \( e \) if for each \( s \in S_u \), \( sa \in S_v \).

**Definition 2.6.3.** \( S_v \) is backward-consistent if for each \( s \in S_v \) such that \( |s| \geq 1 \), there is a vertex \( u \) and an edge \((u, v)\) labeled \( a \) for some \( a \in \Sigma \), with the property that \( s = s'a \) and \( s' \in S_u \).

**Definition 2.6.4.** The specifications for a finite automata are consistent if

1. The specifications are forward-consistent with respect to every edge.
2. The specifications are backward-consistent.
3. \( \lambda \in S_{\text{start}} \) and \( \lambda \notin S_v \) for \( v \neq \text{start} \), where start denotes the start vertex.
Lemma 2.6.5. The specifications for a finite automata with no \(\lambda\)-edges are consistent exactly if they specify the strings that can reach each vertex.

Proof. Suppose for a contradiction that the specifications are consistent yet some specification is incorrect, i.e. it fails to specify exactly those strings that can reach its vertex.

Case 1. There is a shortest string \(s\) that is incorrectly included in some specification \(S_u\).

By (3), \(s \neq \lambda\). So \(s\) can be written as \(s = s'a\), where \(a \in \Sigma\). By backward-consistency, \(s' \in S_u\) for some vertex \(u\) where \((u, v)\) is an edge labeled \(a\). Now \(s'\) is correctly included in \(S_u\) as \(s\) was a shortest incorrectly placed string; but as \(s'\) is correctly in \(S_u\), then in fact \(s\) is correctly in \(S_u\). So this case does not arise.

Case 2. There is a shortest string \(s\) that is incorrectly missing from some specification \(S_u\).

Again, \(s \neq \lambda\). So \(s\) can be written as \(s = s'a\), where \(a \in \Sigma\). Let \(U\) be the set of vertices with an edge labeled \(a\) into \(v\) (i.e. for each \(u \in U\), there is an edge \((u, v)\) labeled by \(a\)). Then \(s'\) reaches some vertex in \(U\), vertex \(u\) say. As \(|s'| < |s|\), \(s' \in S_u\). But then \(s \in S_u\) by forward consistency. So this case does not occur either.

We conclude that if the specifications are consistent then all the specifications are correct.

To show that specification correctness implies consistency just entails checking that Conditions 1–3 above hold. This is immediate from the definitions of forward and backward consistency and is left to the reader. \(\Box\)

We account for the \(\lambda\)-edges by making appropriate small changes to the definition of consistency.

What we want to do is to modify the automata by removing all \(\lambda\)-edges and introducing appropriate replacements. Let \(M\) be an NFA; we build an equivalent NFA \(M'\) with no \(\lambda\) edges, as follows.

1. Make all vertices reachable on input \(\lambda\) into start vertices.

2. For each pair of vertices \((u, v)\) and for each label \(a \in \Sigma\), if there is path from \(u\) to \(v\) labeled by \(a\), introduce an edge \((u, v)\) labeled by \(a\), and then remove all \(\lambda\)-edges.

We need to explain what it means to have more than one start vertex: \(M'\) is simply understood as starting at all its start vertices simultaneously.

It is not hard to see that \(M'\) recognizes exactly the same strings as the original machine. The result of Lemma 2.6.5 applies to \(M'\). Note that the vertex specifications used for \(M\) are the ones being used for \(M'\) too. Consequently, one can check the correctness of the specifications for \(M\) by checking that they are consistent.

Exercises

1. Consider the Finite Automata shown in Figure 2.35.

   Answer the following questions regarding each of the automata.

   i. Name its start vertex.

   ii. List its recognizing vertices.

   iii. List the series of vertices the finite automata goes through as the following input is read: \(ababb\).

   iv. Is \(ababb\) recognized by the automata?
2. Draw the graphs of Deterministic Finite Automata recognizing the following languages. In each subproblem, the alphabet being used is $\Sigma = \{a, b\}$.

i. The empty set $\emptyset$.

ii. The set containing just the empty string: $\{\lambda\}$.

iii. The set of all strings: $\Sigma^*$.

iv. The set of all strings having at least one character: $\{w \mid |w| \geq 1\}$.

**Sample solution.**

v. The set of all strings of length two or more: $\{w \mid |w| \geq 2\}$.

vi. The set of all strings that begin with an $a$.

vii. The set of all strings that end with a $b$.

viii. The set of all strings containing $aa$ as a substring.

ix. The set of all strings containing at least four $a$'s.

x. The set of all strings either starting with an $a$ and ending with a $b$, or starting with a $b$ and ending with an $a$.

xi. The set of all strings such that no two $b$'s are adjacent.

xii. The set of all strings excepting $aba$: $\{w \mid w \neq aba\}$.

xiii. The set of all strings with alternating $a$'s and $b$'s.

xiv. The set of all strings with $a$'s in all the even positions and $a$'s or $b$'s in the odd positions (so $\lambda, bab, aab, b$ are in this set, but $ab$ is not).

xv. The set of all strings of even length whose second symbol is a $b$.

xvi. The set of all strings containing at least two $a$'s.
xvii. The set of all strings that contain \textit{aba} as a substring.

xviii. The set of strings in which all the \textit{a}'s come before all the \textit{b}'s.

3. Consider the construction from Section 2.2.4. Show how to modify this construction so as to give a (Deterministic) Finite Automata \( M \) recognizing \( A \cap B \), assuming that \( A \) and \( B \) can both be recognized by Finite Automata. Explain briefly why your construction is correct; that is, explain why \( L(M) = A \cap B \).

4. Each of the following languages can be obtained by applying set operations (one of union, intersection, or complement) to simpler languages. By building Deterministic Finite Automata recognizing the simpler languages and then combining or modifying them, build Deterministic Finite Automata to recognize the following languages. For intersections, Problem 3 may be helpful. In each subproblem, the alphabet being used is \( \Sigma = \{a,b\} \).

i. The set of all strings that contain at least one \textit{a} or at least two \textit{b}'s.

ii. The set of all strings that do not contain the substring \textit{aa}.

iii. The set of all strings other than the empty string: \( \{w \mid w \neq \lambda\} \).

iv. The set of all strings other than \textit{a} or \textit{bb}: \( \{w \mid w \neq a, bb\} \).

v. The set of all strings containing at least one \textit{a} and at least one \textit{b}.

vi. The set of all strings containing at least one \textit{a} and at most two \textit{b}'s.

vii. The set of all strings with an even number of \textit{a}'s and an odd number of \textit{b}'s.

viii. The set of all strings with an even number of \textit{a}'s and no adjacent \textit{a}'s.

ix. The set of all strings that start with an \textit{a} and end with the character \textit{b}.

x. The set of all strings that have either 2 or 3 \textit{b}'s and that have exactly 2 \textit{a}'s.

5. Draw the graphs of NFAs recognizing the following languages. In each subproblem, the alphabet being used is \( \Sigma = \{a,b\} \).

i. \( L = a^* \), using a 1-vertex NFA. Why is your solution not a DFA?

ii. \( L_2 = \{w \mid w \text{ ends with } bb\} \), using a 3-vertex, 3-edge NFA.

iii. \( L_3 = \{w \mid w \text{ has both } aa \text{ and } bb \text{ as substrings}\} \), using a 9-vertex,13-edge NFA (8 vertices and 12 edges is doable).

iv. \( L_4 = \{w \mid w \text{ is of even length or the second symbol in } w \text{ is a } b \text{ (or both)}\} \), using a 5-vertex NFA.

6. Using the method of Section 2.3.2, convert the following NFAs to DFAs.

i. The solution to Problem 5(i).

ii. The solution to Problem 5(ii).

iii. The solution to Problem 5(iv).

iv. The NFA in Figure 2.36(a).

v. The NFA in Figure 2.36(b).
7. Let the following languages all be subsets of \( \{a, b\}^* \):

- \( A = \{ w \mid w \text{ begins with an } a \} \).
- \( B = \{ x \mid x \text{ ends with the character } b \} \).
- \( C = \{ w \mid |w| \geq 2 \} \).
- \( D = \{ x \mid x \text{ contains } aa \text{ as a substring} \} \).
- \( E = \{ w \mid \text{all characters in even positions in } w \text{ are } a\text{'s and } w \text{ has even length} \} \), i.e. the second, fourth, sixth, etc. characters are all \( a\text{'s} \).
- \( F = \{ w \mid w \text{ contains an } a \} \).
- \( G = \{ w \mid w \text{ begins with a } b \} \).
- \( H = \{ w \mid w \text{ contains } ba \text{ as a substring} \} \).
- \( J = \{ w \mid w\text{'s next to last character is an } a \} \).
- \( K = \{ w \mid \text{all characters in odd positions in } w \text{ are } a\text{'s and } w \text{ has even length} \} \). (See the definition of \( E \) also.)

Using the methods of Section 2.3.1, give the graphs of NFAs that recognize the following languages.

i. \( A \cup B \).
ii. \( F \cup G \).
iii. \( C \circ D \).
iv. \( H \circ J \).
v. \( E^* \).
vi. \( K^* \).
vii. \( L = \phi^* \) (recall that \( \phi \) is the empty language). Trust the construction. What strings, if any, are in \( \phi^* \)?
8. For each of the following languages, (a) construct an NFA recognizing the language, and then (b) convert the NFA to a DFA recognizing the same language using the method of Section 2.3.2. You need show only the portion of the DFA reachable from the start vertex.

   i. \(K = \{ba, bab\}^*\). The NFA must have a single recognizing vertex, and 4 vertices altogether.
   ii. \(L = \{b, bab\}^*\). The NFA must have a single recognizing vertex, and 4 vertices altogether.

9. Let \(L\) be a regular language. Define the reverse of \(L\), \(L^R = \{w \mid w^R \in L\}\), i.e. \(L^R\) contains the reverse of strings in \(L\). Show that \(L^R\) is also regular.
   Hint. Suppose that \(M\) is a DFA (or an NFA if you prefer) recognizing \(L\). Construct an NFA \(M^R\) that recognizes \(L^R\); \(M^R\) will be based on \(M\). Remember to argue that \(L(M^R) = L^R\).

10. Give regular expressions that represent the following languages. Suggestion: Test your solutions with a few strings both in and not in the language.

   i. \(A = \{w \mid w\) has an even number of b's\}.
   ii. \(B = \{w \mid \) every a in w is preceded by a b\}.
   iii. \(C = \{w \mid w\) does not contain the substring bb\}.
   iv. \(D = \{w \mid \) the third to last character in w is a b\}.
   v. \(E = \{w \mid w\) contains at least two b's\}.

   *vi. Let \(\Sigma = \{0, 1, 2\}\. Let \(F = \{w \mid \) the sum of the digits in w mod 3 equals 0\}. Give a direct construction without using Lemma 2.5.4. Explain why your solution is correct.

11. For each of the following regular expressions answer the following questions:

   i. \((a \cup b)^*a(a \cup b)^*\).
   ii. \((a \cup b)^*a(a \cup b)^*b(a \cup b)^*\).
   iii. \((\lambda \cup a)b^*\).
   iv. \((\lambda \cup aa)b^*\).

   a. Give two strings in the language represented by the regular expression.
   b. Give two strings not in the language represented by the regular expression.
   c. Describe in English the language represented by the regular expression.

12. Using the method of Lemma 2.5.1, give NFAs that recognize the languages represented by the following regular expressions.

   a. \((a \cup bb)^*(aba)\).
   b. \((bab)(a \cup bb)^*\).
   c. \(((ab)^* \cup a)b\).

13. Using the method of Lemma 2.5.4, give regular expressions representing the languages recognized by the following NFAs.
14. Use the Pumping Lemma to show that the following languages are not regular.

i. \( L = \{ a^i b^i \mid i \geq 0 \} \).

**Sample solution.** Suppose, so as to obtain a contradiction, that \( L \) were regular. Then \( L \) must have a pumping length \( p \geq 1 \) such that for any string \( s \in L \) with \( |s| \geq p \), \( s \) can be pumped.

Choose \( s \) to be the string \( s = a^p b^p \).

As \( s \in L \) and \( |s| = 2p \geq p \), \( s \) can be pumped. In other words, we can write \( s = xyz \), with \( |y| > 0 \), \( |xy| \leq p \), and \( xy^i z \in L \) for every integer \( i \geq 0 \).

By pumping \( s \), we will obtain a contradiction, as follows.

As the first \( p \) characters of \( s \) are all \( a \)'s, the substring \( xy \) must also be a string of all \( a \)'s. As \( xy^i z \in L \) for any integer \( i \geq 0 \), on setting \( i = 0 \), we have that \( xz \in L \); but the string \( xz \) has removed \( |y| \) \( a \)'s from \( s \), that is \( xz = a^{p-|y|} b^p \), and this string is not in \( L \) since \( p - |y| \neq p \). This is a contradiction for we have shown that \( xz \in L \) and \( xz \notin L \).

Consequently the initial assumption is incorrect; that is, \( L \) is not regular.

ii. \( A = \{ a^i b a^i \mid i \geq 0 \} \).

iii. \( B = \{ a^{2i} b^i \mid i \geq 1 \} \).

iv. \( C = \{ a^i b^i c^i \mid i \geq 1 \} \).

v. Let \( \Sigma = \{ (, ) \} \) and let \( D \) be the language of legal balanced parentheses: i.e., for each left parenthesis there is a matching right parenthesis to its right, and pairs of matched parentheses do not interleave. For example, the following are in \( D \): \( ((), (())(), (((())))() \), and the following are not in \( D \): \( (), (())(), ()( \).

vi. \( E = \{ w w \mid w \in \{ a, b \}^* \} \).

vii. \( F = \{ w \mid w = w^R \} \). Such \( w \) are called **palindromes**.

viii. \( G = \{ w w^R w \} \).

ix. \( H = \{ a^{2^i} \mid i \geq 0 \} \), strings with \( a \) repeated \( 2^i \) times for some \( i \).
x. \( I = \{a^q \mid q \text{ prime}\} \), strings containing a prime number of \( a \)'s.

15. If for language \( L \) the pumping length is 4 and \( a^4b^4 \in L \), then if you choose \( s = a^4b^4 \) and apply the Pumping Lemma to \( s \), how many pairs \( x, y \) are there, and what are these pairs?

16. Does the proof of the Pumping Lemma work using a \( p \)-state NFA accepting regular language \( L \) instead of a \( p \)-state DFA? Justify your answer briefly.

17. Let \( A = \{a^i w \mid w \in \{a, b\}^* \text{ and } w \text{ contains at most } i \text{ a's, } i \geq 1\} \). Show that \( A \) is not regular. Remember that if \( R \) is regular and if \( A \) is regular then so is \( A \cap R \). It may be helpful to choose a suitable \( R \), and then show that \( A \cap R \) is not regular.

18. Show that the following languages are not regular.

   i. \( A = \{a^i b^j \mid i \neq j\} \).

   ii. \( B = \{u v v u \mid u, v \in \{a, b\}^* \text{ and } v \in a^*\} \).

   iii. \( C = \{u v u \mid u, v \in \{a, b\}^* \text{ and } u, v \neq \lambda\} \).

   iv. \( D = \{w \mid w \neq w^R\} \).

19. Use the Pumping Lemma to show that the following languages are not regular.

   i. \( A = \{m = n * r \mid m, n, r \in \{0, 1\}^* \text{ and } m = n * r\} \).

      The strings in \( A \) include the equal sign and the multiplication sign.

   ii. \( B = \{m = n + r \mid m, n, r \in \{0, 1\}^* \text{ and } m = n + r\} \).

      The strings in \( B \) include the equal sign and the addition sign.

20. Let \( C = \text{shuffle}(A, B) \) denote the shuffle \( C \) of two languages \( A \) and \( B \); it consists of all strings \( w \) of the form \( w = a_1 b_1 a_2 b_2 \cdots a_k b_k \), for \( k > 0 \), with \( a_1 a_2 \cdots a_k \in A \) and \( b_1 b_2 \cdots b_k \in B \). Show that if \( A \) and \( B \) are regular then so is \( C = \text{shuffle}(A, B) \).

21. Define \( \text{ROC}(w) \) to be the collection of strings obtained from \( w \) by removing exactly one of \( w \)'s characters.

   e.g. \( \text{ROC}(bab) = \{ab, bb, ba\} \). Let \( \text{ROC}(L) = \{\text{ROC}(w) \mid w \in L\} \).

   Show that if \( L \) is regular then so is \( \text{ROC}(L) \).

   **Sample solution.** Let \( M \) be a DFA recognizing \( L \). We will construct NFA \( N \) to accept \( \text{ROC}(L) \). The intuitive idea is that \( N \) will be obliged to follow an edge in \( M \) without reading a character exactly once. This is implemented as follows. The graph for \( N \) consists of two copies of the graph for \( M \), with the first copy joined to the second by \( \lambda \)-labeled edges. Hence a path in \( N \) transits once from the first copy to the second. Having the start vertex for \( N \) in the first copy and the recognizing vertices in the second copy ensures that a string is recognized exactly if it omits one character that would be read on the corresponding path in \( M \).

   The details follow. \( N \) comprises two copies of \( M \), named \( M_1 \) and \( M_2 \). For each vertex \( p \) in \( M \) there will be a vertex \( p_1 \) in \( M_1 \) and a vertex \( p_2 \) in \( M_2 \). And for each \( a \)-edge \((p, q)\) in \( M \) (an
a-edge is an edge labeled by a), in \( N \) there will be a-edges \((p_1, q_1)\) and \((p_2, q_2)\) plus a \(\lambda\)-edge \((p_1, q_2)\). We will say that edge \((p_1, q_2)\) has pseudo-label \(a\), meaning that \((p_1, q_2)\) was obtained from a-edge \((p, q)\). An edge \((p_1, q_2)\) could have more than one pseudo-label. Finally, for each recognizing vertex \(r \in M\) there will be a recognizing vertex \(r_2 \in N\), and if \(s\) is the start vertex for \(M\), \(s_1\) will be the start vertex for \(N\).

Suppose \(w\) is recognized by \(M\). Let \(w = uav\) for some character \(a \in \Sigma\) and strings \(u, v \in \Sigma^*\) (so \(w \neq \lambda\)). Then there is a recognizing path \(P\) for \(w\) which can be partitioned into path \(P_u\) labeled by \(u\), followed by an \(a\)-edge \((p, q)\), followed by a path \(P_v\) labeled by \(v\). Consider the following path \(P'\) in \(N\): It comprises path \(P_u\) in \(M_1\), followed by edge \((p_1, q_2)\), followed by path \(P_v\) in \(M_2\). Clearly \(P'\) is a recognizing path in \(M'\) and \(P'\) has label \(u\lambda v = uv\). This construction works for any of the characters in \(w\), and thus the resulting strings \(w' = uv\) form the set \(\text{ROC}(w)\), which is therefore recognized by \(N\). As this applies to any \(w\) recognized by \(M\), it follows that \(\text{ROC}(L) \subseteq L(N)\).

On the other hand, if \(w\) is recognized by \(N\), then there is a recognizing path \(P'\) for \(w\) which comprises a path \(P_u\) in \(M_1\), followed by an edge \((p_1, q_2)\), followed by a path \(P_v\) in \(M_2\). Let \(u\) be the label on \(P_u\), \(v\) the label on \(P_v\), and \(a\) the pseudo-label on \((p_1, q_2)\) ((or one of them if there are several). Consider the path \(P\) in \(M\) comprising \(P_u\), then edge \((p, q)\), then path \(P_v\). It has label \(w = uav\) and is recognizing. Clearly \(w' \in \text{ROC}(w)\). Thus for each \(w' \in L(N)\) there is a \(w \in L(M)\) with \(w' \in \text{ROC}(w)\). In other words \(L(N) \subseteq \text{ROC}(L)\).

This shows that \(N\) recognizes \(\text{ROC}(L)\), and hence \(\text{ROC}(L)\) is regular if \(L\) is regular.

22. Let \(w' = \text{Subst}(a, b, w)\), or \(\text{Subst}(a, b, w)\) for short, if \(w'\) is obtained by replacing every \(a\) in \(w\) by \(b\). For example, \(\text{Subst}(a, b, caca) = cbcb\).

Let \(\text{Subst}(a, b, L) = \{w' \mid w' = \text{Subst}(a, b, w)\text{ for some }w \in L\}\). Show that if \(L\) is regular then so is \(\text{Subst}(a, b, L)\).

23. Let \(w = w_1w_2 \cdots w_k\) where each \(w_i \in \Sigma, 1 \leq i \leq k\). And let \(h\) be a function from the alphabet \(\Sigma\) into the set of strings \(\Gamma^*\), so that \(h(a) \in \Gamma^*\) for each \(a \in \Sigma\).

Define \(\text{FullSubst}(w, h)\), or \(\text{FS}(w, h)\) for short, to be \(h(w_1)h(w_2) \cdots h(w_k)\). Note that \(\text{FullSubst}(\lambda, h) = \lambda\).

e.g. if \(h(a) = cc, h(b) = de\), then \(\text{FS}(aba) = ccdecc\).

For \(L \subseteq \Sigma^*\), define \(\text{FS}(L, h) = \{\text{FS}(w, h) \mid w \in L\}\).

Show that if \(L\) is regular then so is \(\text{FS}(L, h)\).

24. Let \(w = w_1w_2 \cdots w_k\) where each \(w_i \in \Sigma, 1 \leq i \leq k\). And let \(r\) be a function mapping characters in \(\Sigma\) into regular expressions over the alphabet \(\Gamma\).

Define \(\text{SubstitRegExpr}(w, r)\), or \(\text{SRE}(w, r)\) for short, to be \(r(w_1)r(w_2) \cdots r(w_k)\). Note that \(\text{SRE}(\lambda, r) = \lambda\).

e.g. if \(r(a) = c^*\), \(r(b) = \lambda \cup d\), then \(r(ab) = c^*(\lambda \cup d)\).

For \(L \subseteq \Sigma^*\), define \(\text{SRE}(L, r) = \{\text{SRE}(w, r) \mid w \in L\}\).

Show that if \(L\) is regular then so is \(\text{SRE}(L, r)\).

25. Define \(\text{RemoveOneSymbol}(w, a)\), or \(\text{ROS}(w, a)\) for short, to be the string \(w\) with each occurrence of \(a\) deleted.

e.g. \(\text{ROS}(abac, a) = bc\), \(\text{ROS}(bc, a) = bc\).
Let $\text{ROS}(L, a) = \{ \text{ROS}(w, a) \mid w \in L \}$. Show that if $L$ is regular then so is $\text{ROS}(L, a)$.

26. Define $\text{ReplaceOneSymbol}(w, a, b)$, or $\text{RpOS}(w, a, b)$ for short, to be the collection of strings obtained from $w$ by replacing exactly one occurrence of $a$ with $b$.

- e.g. $\text{RpOS}(bb, a, b) = \{ \}$; $\text{RpOS}(acca, a, b) = \{ bcba, accb \}$.
- Let $\text{RpOS}(L, a, b) = \{ \text{RpOS}(w, a, b) \mid w \in L \}$.
- Show that if $L$ is regular then so is $\text{RpOS}(L)$.

27. The next problem introduces yet another way of defining regular languages.

i. Let $L$ be a language over the alphabet $\Sigma$. Let $R(x, y)$ be the following property defined with respect to $L$: $R(x, y)$ is TRUE exactly if for every string $z \in \Sigma^*$, $xz \in L \iff yz \in L$.

Prove that $R$ is an equivalence relation, that is that: (a) $R(x, x)$ is TRUE, (b) $R(x, y)$ being TRUE implies $R(y, x)$ is also TRUE, and (iii) if $R(x, y)$ and $R(y, z)$ are both TRUE, then so is $R(x, z)$. We write $x \equiv y$ if $R(x, y)$ is TRUE, or sometimes $x \equiv_L y$ to emphasize the dependence on the language $L$.

ii. $X$ is an equivalence class with respect to $R$ if (a) for every pair of strings $x, y \in X$, $x \equiv y$, and (b) for every $x \in X$, for every $y$ such that $x \equiv y$, $y \in X$ also.

Suppose that there is a DFA $M$ recognizing $L$.

- (a) In addition, suppose that input strings $x$ and $y$ have the same destination vertex $p$ in $M$.
- Then show that $x \equiv y$.

- (b) Conclude that $L$ has a finite number of equivalence classes.

iii. Show that if $R$ partitions $\Sigma^*$ into a finite number of equivalence classes then there is a DFA recognizing $L$.

Hint: What is the first thing to try as possible descriptors for the vertices in the DFA?

This question shows that a language is regular if and only if it has a finite number of equivalence classes.

28. Let $\text{Prefix}(L) = \{ u \mid \text{there is a string } v \text{ such that } uv \in L \}$. Show that if $L$ is regular then so is $\text{Prefix}(L)$.

Hint: Let $u$ be a prefix of $w$ (i.e. there is a string $v$ such that $uv = w$). Let $M$ be a DFA recognizing $L$. What can you say about the relationship between the destination vertices reached in $M$ by strings $u$ and $w$?

29. Let $\text{Suffix}(L) = \{ v \mid \text{there is a string } u \text{ such that } uv \in L \}$. Show that if $L$ is regular then so is $\text{Suffix}(L)$.

30. Let $\text{Continuation}(L)$ or $\text{Cont}(L)$ for short be defined by $\text{Cont}(L) = \{ w \mid \text{there is a string } u \in L \text{ and a string } v \text{ such that } uv = w \}$.

Show that if $L$ is regular then so is $\text{Cont}(L)$.

31. Let $\text{Inf-Cont}(L) = \{ u \mid u \in L \text{ and for infinitely many } v, uv \in L \}$. Show that if $L$ is regular then so is $\text{Inf-Cont}(L)$.

32. Let $\text{All-Cont}(L) = \{ u \mid \text{for all strings } w, uw \in L \}$. Show that if $L$ is regular then so is $\text{All-Cont}(L)$.

33. Let $\text{All-Pref}(L) = \{ w \mid \text{for all prefixes } u \text{ of } w, u \in L \}$. Show that if $L$ is regular then so is $\text{All-Pref}(L)$. (Recall that $u$ is a prefix of $w$ if there is a string $v$ such that $uv = w$.)
34. i. Let $1/2L = \{u \mid \exists v \text{ with } |v| = |u| \text{ and } uv \in L\}$. 
Suppose that $L$ is regular; then show that $1/2L$ is also regular. 
Hint. Think nondeterministically. You will need to “guess” $v$. Getting its length right can be done only as $u$ is being read. What else do you need to guess, and what needs to be checked?

ii. Let Square-$L = \{u \mid \exists v \text{ with } |v| = |u|^2 \text{ and } v \in L\}$. 
Show that if $L$ is regular then so is Square-$L$.

iii. Let Power-$L = \{u \mid \exists v \text{ with } |v| = 2^{|u|} \text{ and } v \in L\}$. 
Show that if $L$ is regular then so is Power-$L$.

35. i. Let $L = \{a^3, a^5\}$. Show that $L^*$ includes all strings of 15 or more $a$’s. Hence conclude that $L^*$ is regular.

ii. Let $L \subset a^*$ be finite. Show that $L^*$ is regular.

36. Consider the following variant of a DFA, called a *Mealy-Moore machine*, that writes an output string as it is processing its input. Viewed as a graph, it is like a DFA but each edge is labeled with both an input character $a \in \Sigma$, as is standard, and an output character $b \in \Gamma$ or the empty string $\lambda$, where $\Gamma$ is the output alphabet (possibly $\Sigma = \Gamma$).

The output $M(x)$ produced by $M$ on input $x$ is simply the concatenation of the output labels on the path followed by $M$ on reading input $x$. The output language produced by $M$, $O(M)$, is defined to be 

$$O(M) = \{M(x) \mid x \in L(M)\},$$

the output strings obtained on inputs $x$ that $M$ recognizes.

Show that $O(M)$ is regular.

37. A 2wayDFA is a variant of a DFA in which it is possible to go back and forth over the input, with no limit on how often the reading direction is reversed.

This can be formalized as follows. The input $x \in \Sigma^*$ to the DFA is sandwiched between symbols $\dagger$ and $\ddagger$, so the DFA can be viewed as reading string $x_0x_1x_2 \cdots x_nx_{n+1}$, where $x = x_1x_2 \cdots x_n$, $x_0 = \dagger$, and $x_{n+1} = \ddagger$. The DFA is equipped with a *read head* which will always be over the next symbol to be read. At the start of the computation, the read head is over the symbol $x_1$, and the DFA is at its start vertex.

In general, the DFA will be at some vertex $v$, with its read head over character $x_i$ for some $i$, $0 \leq i \leq n+1$. On its next move the DFA will follow the edge leaving $v$ labeled $x_i$. This edge will also carry a label L or R indicating whether the read head moves left (so that it will be over symbol $x_{i-1}$), or right (so that it will be over $x_{i+1}$). The latter is the only possible move for a standard DFA. There are two constraints: when reading $\dagger$ only moves to the right are allowed and when reading $\ddagger$ only moves to the left are allowed. If there is no move, the computation ends.

A 2wayDFA $M$ recognizes an input $x$ if it is at a recognizing vertex when the computation ends. Show that the language recognized by a 2wayDFA is regular.

Hint. Consider the sequence of vertices $M$ is at when its read head is over character $x_i$. Observe that in a recognizing computation there can be no repetitions in such a sequence. Now create
a standard NFA $N$ to simulate $M$. A state or supervertex of $N$ will record the sequence of vertices that $M$ occupies when its read head is over the current input character. The constraints on $N$ are that its moves must be between consistent supervertices (you need to elaborate on what this means). Also, you need to specify what are the recognizing supervertices of $N$, and you need to argue that $M$ and $N$ recognize the same language.