NOTES ON FINITE STATE AUTOMATA

Let’s start by giving the implementation of the Knuth-Morris-Pratt string searching algorithm using a finite state automata.

The Knuth-Morris-Pratt algorithm is a method for searching text for a string. It is particularly good when the text and search strings are binary digits. A naive approach to this problem would be to start at the first text bit seeing if that is the start of the string if not start at the second text bit and so on. This approach may examine the text bits multiple times. For example with the text string 1111100000000 the 6th 1-bit will be examined 6 times when searching for the substring 111111. The Knuth-Morris-Pratt algorithm by contrast examines every text bit just once.

The following finite state automaton (illustrated below) is hardwired to search for the string 101011001011. The finite state automaton works as follows, with each forward step (solid edge) we take the next text bit and compare it with the contents of the box at the right hand end of the edge if there is a match continue on the forward edge otherwise take the backwards (dashed edge) then examine contents of the box (whether we went forwards or backwards) for match with the next text bit etc. If we reach the end marked success we have found the string otherwise if we run out of text to be searched it is not there.

You may like to use the finite state automata to search the text
1100010101010110001010110011101011001011101
for the string 101011001011.

We are going to describe a collection of theoretical machines called finite state automata. These machines are sometimes used as a theoretical model of actual computers. Each such machine is thought of as

---

These lecture notes were compiled in the Department of Mathematics and Statistics in the University of Melbourne for the use of students in the various subjects. Copyright C. F. Miller, J. R. J. Groves, Walter Neumann and David Coulson 1989-2008.

1
NOTES ON FINITE STATE AUTOMATA

reading a string of characters from some fixed alphabet $\Sigma$ as input. As each character is read, the machine changes from one state to another depending on its current state and on the character read from the input. When the input string is exhausted, the machine is said to accept or reject the string depending on what state it finds itself in at the end. Here is a formal definition.

Definition 1. A (deterministic) finite state automaton $M$ is a 5-tuple $(\Sigma, S, \tau, s_0, A)$ which consists of a finite set $\Sigma$ of symbols called the alphabet of $M$, a finite set $S$ called the states of $M$, a function $\tau : S \times \Sigma \to S$ called the transition function, a distinguished element $s_0 \in S$ called the initial state or start state, and a subset $A \subseteq S$ called the set of accept states.

It is usually convenient to think of a finite state automaton $M$ as graph $\Gamma_M$ with labeled, oriented edges as follows. The states of the machine $M$ correspond to the vertices of $\Gamma_M$, so we identify states with their corresponding vertices. The edges of $\Gamma_M$ are defined by the rule that if $s_j = \tau(s_i, a_m)$ then there is an oriented edge labeled by $a_m$ from $s_i$ to $s_j$. We adopt the pictorial convention of labeling the vertices of $A$ (which are accept states) by small circles, and those which are not in $A$ (fail states) by solid dots of the same size. We also always label the start state $s_0$. In some cases, when we do not label the graph with the names of the states, we will use an arrow to point to the initial state. With these conventions, the graph $\Gamma_M$ completely determines $M$ and vice versa.

To illustrate the concepts of the definition look back at the hard wired FSA illustrated previously, we see that this FSA has 13 states (the 12 boxes and the “success” state. Of these 13 states only the “success” state is accept. The alphabet is $\Sigma = \{0, 1\}$ (usually this is given more explicitly). The transition function is given implicitly by the arrows. In more detail if we were to label the states from left to right $s_0, s_1, \ldots, s_{12}$ (“success” state) then for instance a “0” in the 6’th box $(s_5)$ means we go to the 4’th box so

\[
\begin{align*}
\tau(s_5, 0) &= s_3 \\
\tau(s_5, 1) &= s_6 \\
\tau(s_{10}, 0) &= s_0
\end{align*}
\]

The FSA is a full deterministic FSA as each state has two edges leading out corresponding to each letter of the alphabet $\Sigma$ (excepting “success” where one could imagine both ’0’ and ’1’ edges returning to “success”).

A finite state automaton $M$ is thought of as performing a computation on any word $w \in \Sigma^*$ (the set of all words on the alphabet $\Sigma$) to determine whether or not $w$ lies in a certain subset $L(M)$ of $\Sigma^*$ in the following way. If $w = a_1a_2 \ldots a_k \in \Sigma^*$, the computation begins in the state $s_0$; so the state after reading 0 letters is $s(0) = s_0$. Now $M$ reads
a_1 and this causes \( M \) to enter state \( s(1) = \tau(s_0, a_1) \). \( M \) continues in this way reading the letters of \( w \) in order and changing states according to the rule \( s(j) = \tau(s(j - 1), a_j) \). After reading all \( k \) of the symbols of \( w \) and hence ending in state \( s(k) \), the machine \( M \) concludes that \( w \in L(M) \) if and only if \( s(k) \in A \), that is, exactly when it finishes in an accept state.

The set of words \( L(M) \) thus defined is called the language of \( M \) or the language accepted by \( M \). A language is said to be a regular language if it is the language accepted by some finite state automaton.

So the language of the hard wired FSA (illustrated above) is any binary string containing the substring 101011001011.

The computation of a machine \( M \) with an input word \( w \) corresponds to traversing a path in \( \Gamma_M \) starting at \( s_0 \). As successive symbols of \( w \) are read, we traverse the oriented edges of \( \Gamma_M \) with corresponding labels. The word \( w \) is in \( L(M) \) if and only if this path ends in an accept state.

Here is an example. Consider the language consisting of the set of words \( \{a^n b^m \mid n, m \geq 0\} \), that is words consisting of 0 or more occurrences of \( a \) followed by 0 or more occurrences of \( b \). We will show this is a regular language by producing a finite state automaton \( M \) whose language \( L(M) \) is exactly this set of words.

We take as alphabet \( \Sigma = \{a, b\} \) and as set of states \( S = \{s_0, s_1, s_2\} \) and as accept states and \( A = \{s_0, s_1\} \). The start state is as usual \( s_0 \).

The transition function is then defined by the following table which shows the effect of the transition function \( \tau \) applied to the corresponding row (indexed by a state) and column (indexed by a letter).

<table>
<thead>
<tr>
<th>( \tau )</th>
<th>( a )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>( s_0 )</td>
<td>( s_1 )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( s_2 )</td>
<td>( s_1 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( s_2 )</td>
<td>( s_2 )</td>
</tr>
</tbody>
</table>

This machine \( M \) starts reading a word in state \( s_0 \). As long as only \( a \)'s are read it continues to stay in state \( s_0 \). When the first \( b \) is read it goes to state \( s_1 \). As long as only \( b \)'s are read it continues to stay in state \( s_1 \). If an \( a \) is then read it goes to state \( s_2 \) and stays there no matter what symbols are subsequently read. State \( s_2 \) is a fail state since in order to get there we must have read a word of the form \( a^n b^m a \) where \( n, m \geq 1 \) which is not an initial segment of any word in the desired language. As the only two accept states are \( s_0 \) and \( s_1 \) it is now clear this automata accepts exactly the intended language.

Below is the graph corresponding to this automata. Usually it is easier to see from the graphical version what language is accepted by the machine.
Notice that in the graphical version, at any vertex there is exactly one edge labeled by each symbol departing from that vertex. This property is equivalent to the requirement in the definition that \( \tau \) is a function from \((all of) \ S \times \Sigma \) to \(S\). It is sometimes convenient to consider a partial deterministic automaton \(M\) in which one only requires that \( \tau \) be defined on a subset of \(S \times \Sigma\). Then in the graphical version there is at most one edge in \(\Gamma_M\) labeled by each symbol departing from each vertex.

In case \(M\) is a partial deterministic automaton which is reading a word \(w \in \Sigma^*\) it can happen that the state after reading \(j\) symbols is \(s(j)\) and the next symbol is \(a_i\) and \(\tau(s(j), a_i)\) is not defined. Equivalently, the path in \(\Gamma_M\) has been traversed to state \(s(j)\) and there is no departing arrow labeled \(a_i\) where \(a_i\) is the next unread symbol in \(w\). In this case the machine \(M\) is to conclude that \(w \notin L(M)\) the language accepted by \(M\). Otherwise the definition of \(L(M)\) is as before.

It is often easier to specify a partial deterministic automaton which accepts a particular language because fewer arrows are required. For instance, the language \(\{a^n b^m | n, m \geq 0\}\) in the above example is the language of the partial deterministic automaton with the following graph.

If a partial deterministic automaton \(M\) is not a (deterministic) finite state automaton, it is easy to convert it into a finite state automaton \(M_1\) with \(L(M_1) = L(M)\) in the following way: \(M_1\) is obtained by adding a single new (fail) state \(s_\infty\) to \(M\) and for each state \(s_j\) and symbol \(a_i\) such that \(\tau(s_j, a_i)\) is undefined, in \(M_1\) put \(\tau(s_j, a_i) = s_\infty\); that is, connect \(s_j\) to \(s_\infty\) by a directed edge labeled by \(a_i\). It is clear that the resulting \(M_1\) is a finite state automaton and that \(L(M_1) = L(M)\).

As another example consider the language \(\{a^n b^m | n, m \geq 1\}\). A partial deterministic automaton which accepts exactly this language is
specified by the following graph. Notice that $s_2$ is the only accept state in this instance.

One might ask about the possibility of allowing several edges with the same label to depart for different destinations. Also one might allow changing to certain other states without reading any symbols. For many purposes it is convenient to have such a more general notion of a machine called a non-deterministic finite state automaton. To define such machines we introduce a new symbol $\epsilon$ which is interpreted as the empty word. We always assume $\epsilon$ does not belong to the alphabet $\Sigma$.

The definition is similar to those above except that the transition function $\tau$ is now taken to be a function from a subset of $S \times (\Sigma \cup \{\epsilon\})$ to the set of subsets of $S$ denoted $\text{Pow}(S)$ (this means we may have multiple arrows with the same label exiting a state), that is the value of $\tau$ is a subset of the states - namely those to which the given state is to be connected by appropriately labeled arrows. The definition of the accepted language $L(M)$ of a non-deterministic finite state automaton $M$ is a bit more complicated. If $w = a_1a_2\ldots a_k$ is a word then $M$ accepts $w$ if there is a path in $\Gamma_M$ starting at $s_0$ traversing edges labeled by $a_1$ to $a_k$ with possible traversing of edges labeled $\epsilon$ interspersed. That is we must walk along a path labeled by $w$ except that at any stage we may go along an edge labeled by $\epsilon$ before resuming our walk along $w$. (This amounts to changing state without reading a symbol.)

Here is an example of a non-deterministic finite state automaton whose accepted language is all those words on $a$ and $b$ which contain at least two successive $a$'s or two successive $b$'s. The machine must make “choices” on reading the input.

It turns out (surprisingly) that non-deterministic finite state automata are no more powerful than deterministic finite state automata in the sense that they accept the same collection of languages. To help us prove this, it is convenient to introduce the following notion. If $\Sigma$
is an alphabet and \( L \subseteq \Sigma^* \) is a language and \( w \) is a word on \( \Sigma \), then the cone of \( w \) with respect to \( L \) is the set endings \( v \) so that prefix \( w \) together with ending \( v \) gives a word \( wv \) in the language \( L \). In set notation

\[
C_w(L) = \{ v \in \Sigma^* \mid wv \in L \}.
\]

Observe that if \( w \) is not an initial segment of some word in \( L \) then \( C_w(L) = \emptyset \) (the empty set). Also if \( L = \Sigma^* \) then there is only one cone which is just \( \Sigma^* \) itself. We will continue to use \( \epsilon \) to denote the empty word.

Consider our example \( L = \{ a^n b^m \mid n, m \geq 0 \} \). Observe that \( C_{\epsilon}(L) = C_{a}(L) = \{ b^m \mid m \geq 0 \} \) and that \( C_{ba}(L) = \emptyset \). Moreover, these are the only possible cones for words with respect to \( L \) in the sense that any other cone is equal to one of these three as a set of words.

Before outlining some theoretical results we will look at 2 examples of how cones can be used to build a FSA for a given language (if that language is regular).

In the first example we will look at a language with a finite number of words. Let \( \Sigma = \{ a, b \} \) and \( L = \{ ba, bb, aba, abba \} \).

We will use a search tree to find all the cones systematically. We start with the empty prefix examine this then examine the following two prefixes obtained (by adding a and b). We continue doing this for all prefixes except we need not examine a prefix that has a cone the same as that of a previously examined prefix (because we will obtain exactly the same results). Thus we continue until there are no unexamined cones remaining.

Remark: What we are essentially doing is examining all the prefixes in standard order (see sets, functions and relations). For those with a computer science background we are moving through the search tree (in this case binary as \( | \Sigma | = 2 \)) in a breadth first fashion.

Here is a list of the cones we obtain in this example - there are 7 different cones.

1. \( C_{\epsilon}(L) = L = \{ ba, bb, aba, abba \} \), as is the case for any language \( L \). Clearly the endings to be added to the empty word \( \epsilon \) to get words in the language \( L \) are the words in \( L \).
2. \( C_{a}(L) = \{ ba, bba \} \), as \( aba, abba \) are the only two words in \( L \) starting with \( a \).
3. \( C_{b}(L) = \{ a, b \} \), as \( ba, bb \) are the only two words in \( L \) starting with \( b \).
4. \( C_{aa}(L) = \emptyset \) (the empty set) there are no words in \( L \) starting with \( aa \) (and hence for any \( v \in \Sigma^* \), \( C_{av}(L) = \emptyset \).)
5. \( C_{ab}(L) = \{ a, ba \} \), as \( aba, abba \) are the only two words in \( L \) starting with \( ab \).
6. \( C_{ba}(L) = \{ \epsilon \} \), similarly for any nonempty \( v \in \Sigma^* \) \( C_{bav}(L) = \emptyset \).
\( C_{bb}(L) = \{ \epsilon \} = C_{ba}(L) \).
\( C_{aba}(L) = \{ \epsilon \} = C_{ba}(L) \).

7. \( C_{abb}(L) = \{ a \} \), as \( abba \) is the only word in \( L \) starting with \( abb \).
\( C_{aba}(L) = \{ \epsilon \} = C_{ba}(L) \).
\( C_{abbb}(L) = \phi = C_{aa}(L) \).

These cones may be put together into the following FSA. The cone \( C_a(L) \) is reached from \( C_\epsilon(L) \) using an \( a \) edge (just as \( a \) is obtained by adding an \( a \) to the emptyword \( \epsilon \)) other edges in the FSA are obtained likewise. Let us consider 2 further edges. From the cone \( C_{aa}(L) \) which is empty any edge goes to \( C_{aa}(L) \) (as any subsequent prefix gives no words in the language). From cone \( C_b(L) \) a \( b \) would send us to \( C_{bb} \) but this cone has been identified with \( C_{ba} \) thus we have an edge (labelled \( b \)) from \( C_b(L) \) to \( C_{ba} \).

It only remains to decide which states are accept and which are non-accept. This is quite easy a cone corresponds to an accept state if and only if the prefix (of the cone) is in the language (if and only if \( \epsilon \) is in the cone – as a suffix).

In the second example we will look at a language with an infinite number of words. Let \( \Sigma = \{ a, b \} \) and \( L = \{ u(abba)v \mid u, v \in \Sigma^* \} \), that is, all words containing \( abba \) as a subword.

Here is a list of the cones in this example - there are 5 different cones.

1. \( C_\epsilon(L) = L \).
2. \( C_a(L) = \{ bba(\Sigma)^* \} \cup L \), \( a \) followed by \( bba \) then followed by anything else gives a word containing \( abba \) and hence is in the language. Alternatively if \( a \) is followed by a word containing \( abba \) this results in a word in \( L \).
\( C_b(L) = L \), \( b \) starting with a \( b \) is no use in obtaining \( abba \) as a subword.
\( C_{aa}(L) = C_a(L) \), as only the second \( a \) in \( aa \) helps us getting the subword \( abba \).
3. \( C_{ab}(L) = \{ ba(\Sigma)^* \} \cup L \), similar reasoning as for \( C_a(L) \).
\( C_{aba}(L) = C_a(L) \), same reasoning as for \( C_{ab}(L) \).
4 $C_{ab}(L) = \{a(\Sigma)^*\} \cup L$, similar reasoning as for $C_a(L)$.
5 $C_{abba}(L) = \{(\Sigma)^*\}$, any ending will give us a word in $L$.
$C_{abbb}(L) = L$, none of $abbb$ helps to get $abba$ as a subword.

These cones may be put together into the following FSA. (Compare this with the hard wired binary string searching FSA given at the start of the chapter.)

Now onto some very useful results concerning FSA.

If $M$ is a (deterministic) finite state automata and if $M$ on reading the two words $w_1$ and $w_2$ finishes them in the same state, then $C_{w_1}(L) = C_{w_2}(L)$ where $L = L(M)$. Hence the number of different cones of $L(M)$ is at most the number of states of $M$. The situation for non-deterministic machines is more complicated since reading a word can lead to any of a set of states.

The following is the result we want to prove.

**Theorem 1.** The following conditions on a language $L \subseteq \Sigma^*$ are equivalent:

1. $L$ is a regular language, that is $L$ is the language accepted by some (deterministic) finite state automaton;
2. $L$ is the language accepted by some non-deterministic finite state automaton;
3. there are only finitely many different cones $C_w(L)$ for $w \in \Sigma^*$.

**Proof.** (Sketch) It is clear that $L$ regular implies $L$ is accepted by some non-deterministic finite state automaton (that is (1) $\Rightarrow$ (2)).

Assume that (2) holds, so that $L$ is the language accepted by $M$ which is non-deterministic. Consider any non-empty cone $C_w(L)$ so that for some word $v \in \Sigma^*$ we have $wv \in L$. Then there is a path in $\Gamma_M$ which ends in an accept state after reading $wv$. But after reading only those symbols in $w$ this path leads to any one of some finite set of states, say $\{s(k_1), s(k_2), \ldots, s(k_m)\}$. Suppose $w_1$ is any other word which when read by $M$ can lead to exactly the same finite set of states that $w$ led to. Then $M$ will accept $wv$ if and only if it accepts $w_1v$, so $C_w(L) = C_{w_1}(L)$. Thus any non-empty cone is determined by a (finite) set of states. Since there are only finitely many states, there are only finitely many different sets of states. Hence there are only finitely many different cones.
Finally assume (3), that there are only finitely many cones. We construct a deterministic finite state automaton \( M \) whose accepted language is \( L \). The states of \( M \) are in one to one correspondence with the cones with respect to \( L \). As start state \( s_0 \) we choose the cone \( C_\epsilon(L) \).

If \( C_w(L) \) is a cone, say corresponding to the state \( s_j \), and if \( a \in \Sigma \) then define \( \tau(s_j,a) = s_m \) where \( s_m \) is the state corresponding to \( C_{wa}(L) \).

Observe that if \( C_w(L) = C_u(L) \) then \( C_{wa}(L) = C_{ua}(L) \) so that \( \tau \) is well defined. The accept states are defined by the rule that if \( w \in L \) then the state corresponding to \( C_w(L) \) is an accept state.

One can now easily check that the language accepted by \( M \) is precisely \( L \). This completes the proof.

The proof of this theorem actually yields a lot of additional information. In particular we can deduce the following result.

**Corollary 2.** If \( L \) is a regular language and \( n \) is the number of different cones with respect to \( L \), then there is a deterministic finite state automata \( M \) with \( n \) states such that \( L = L(M) \). If \( M' \) is any other finite state automata with \( L(M') = L(M) \) then \( M' \) has at least \( n \) states. If \( M' \) has exactly \( n \) states, then \( M \) and \( M' \) are the same up to a relabeling of their states.

Here is another useful fact.

**Lemma 3 (Pumping Lemma).** Suppose that \( L \) is a regular language and \( n \) is the number of different cones with respect to \( L \). If \( L \) contains a word \( z \) of length at least \( n \), then \( L \) contains an infinite set of words of the form \( uv^iw \) for all \( i \geq 0 \), where \( z = uvw \) and \( v \) is a non-empty subword of length at most \( n \).

**Proof.** Suppose that \( L \) contains a word \( z \) of length at least \( n \). Let \( M \) be the minimal finite state automaton whose language is \( L \) as above. Then the path in \( \Gamma_M \) corresponding to \( z \) must contain at least \( n \) edges and so must contain a non-trivial loop. Thus \( z = uvw \) where \( v \) corresponds to a non-trivial loop at some state \( s(k) \). Then clearly \( M \) also accepts \( uv^iw \) for all \( i \geq 0 \). This proves the result.

We are now going to give some applications of the above results. First we give an example of a seemingly nice language which is not regular.

**Corollary 4.** The language \( L = \{a^ib^i \mid i \geq 0 \} \) is not regular.

**Proof.** (first version) Observe that \( C_{a^i+b^i}(L) = \{b^i\} \) and so \( L \) has infinitely many different cones. Hence \( L \) is not regular by the above theorem.

(Second version) Suppose \( L \) is regular. Since \( L \) is finite, by the pumping lemma \( L \) contains an infinite set of words of the form \( uv^iw \) for \( i \geq 0 \) where \( v \) is non-empty and \( u \) and \( w \) are fixed. But clearly...
not all of these words can be of the form $a^j b^j$. This is a contradiction, hence $L$ could not be regular. □

Suppose that $L_1$ and $L_2$ are both languages. Their concatenation $L_1 L_2$ is the language $\{uv \mid u \in L_1, v \in L_2\}$.

**Corollary 5.** If $L_1$ and $L_2$ are both regular languages, then their concatenation $L_1 L_2$ is also a regular language.

**Proof.** Let $M_1$ and $M_2$ be finite state automata whose languages are $L_1$ and $L_2$ respectively. We may assume their graphs $\Gamma_{M_1}$ and $\Gamma_{M_2}$ are disjoint. Let $\Gamma$ be the graph formed from the union of these two graphs by connecting every accept state of $\Gamma_{M_1}$ by an $\epsilon$ edge to the start state of $\Gamma_{M_2}$. Then $\Gamma$ is the graph of a non-deterministic finite state automaton with language $L_1 L_2$. Hence by the theorem $L_1 L_2$ is a regular language. □

Here is a simple example of this general result. Let $L_1 = \{(ab)^i \mid i \geq 0\}$ and $L_2 = \{a^{2j} \mid j \geq 0\}$. The following two graphs are partial deterministic automata for these languages respectively.

![Graph 1](image1)

![Graph 2](image2)

The following is then the graph of a non-deterministic finite state automaton $\Gamma$ (as in the above proof) which accepts their concatenation $L_1 L_2$.

![Graph 3](image3)

Here is another easy result with a similar proof.

**Corollary 6.** If $L_1$ and $L_2$ are both regular languages, then their union $L_1 \cup L_2$ is also a regular language.

**Proof.** Let $M_1$ and $M_2$ be finite state automata whose languages are $L_1$ and $L_2$ respectively. We may assume their graphs $\Gamma_{M_1}$ and $\Gamma_{M_2}$ are disjoint. Let $\Gamma$ be the graph formed from the union of these two graphs by adding a new start state $s''_0$ and connecting it to each of the start states of $\Gamma_{M_1}$ and $\Gamma_{M_2}$ by an $\epsilon$ edge. Then $\Gamma$ is the graph of a
non-deterministic finite state automaton with language $L_1 \cup L_2$. Hence by the theorem $L_1L_2$ is a regular language. □

Continuing with the above examples of $L_1$ and $L_2$, the following is then the graph of a non-deterministic finite state automaton $\Gamma$ (as in the above proof) which accepts their union $L_1 \cup L_2$.

The proofs of both of the last two results indicate the usefulness of non-deterministic automata even though they accept the same languages as deterministic automata.

If $L \subseteq \Sigma^*$ is a language, we denote its complement in $\sigma^*$ by $L^c = \Sigma^* \setminus L$.

Lemma 7. If $L$ is a regular language, then its complement $L^c$ is also a regular language.

Proof. Let $M = (\Sigma, S, \tau, s_0, A)$ be a finite state automata whose language is $L$. Then $M^c = (\Sigma, S, \tau, s_0, S \setminus A)$ is a finite state automata whose language is $L^c$. □

(Note that $M^c$ is obtained by just interchanging which states are designated as the fail and accept states.)

Corollary 8. If $L_1$ and $L_2$ are both regular languages, then their intersection $L_1 \cap L_2$ is also a regular language.

Proof. Since $L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$, the corollary follows from the previous two results.

An alternative proof is as follows. Suppose $L_1$ and $L_2$ are both regular languages. A cone $C_w(L_1 \cap L_2)$ with respect to $L_1 \cap L_2$ has the form $C_w(L_1 \cap L_2) = C_w(L_1) \cap C_w(L_2)$. Since $L_1$ and $L_2$ each have only finitely many cones, $L_1 \cap L_2$ can have only finitely many cones. Hence $L_1 \cap L_2$ is also a regular language. This proves the result. □

Recall that we have previously introduce the operation starring operation $^*$ which can be applied to a set of words. So for instance $\{a, bac\}^*$ is the set of all words on $a$ and $bac$ (including the empty word $\epsilon$). It contains $abacbaca$ but not $acbacbac$. Of course, this operation can just as well be applied to infinite sets of words. If $L$ is a language, then $L^*$ is also a language.

Corollary 9. If $L$ is a regular language, then $L^*$ is also a regular language.
Proof. Suppose $L$ is regular so that it is the language of a finite state automaton $M$ having graph $\Gamma_M$. Let $\Gamma$ be the graph formed from $\Gamma_M$ by adding an $\epsilon$ edge from every accept state of $\Gamma_M$ to its start state $s_0$. Then $\Gamma$ is the graph of a non-deterministic finite state automaton with language $L^*$. So by the theorem $L^*$ is a regular language. \hfill \Box

We give a brief sketch of the following result which gives another characterization of regular languages.

**Theorem 10.** Let $\Sigma$ be a finite alphabet. Then any non-empty regular language can be built up from languages of the form $\{x\}$ where $x \in \Sigma \cup \{\epsilon\}$ by a finite sequence of the concatenation, union and starring operations.

Proof. It is convenient to identify automata with their graphs. If $M$ is an automaton and $s_i$ and $s_j$ are two states of $M$, we denote by $L(M, s_i, s_j)$ the language of the machine whose start state is $s_i$ and whose only accept state is $s_j$ and is otherwise the same as $M$ (same states and transition function, that is same edges in its graph).

Let $\Theta$ denote the set of all languages which can be built from the base languages $\{x\}$ using the listed operations. We must show any non-empty regular language belongs to $\Theta$.

It suffices to prove the theorem for languages accepted by partial deterministic automata. If such a machine $M$ has start state $s_0$ and accept states $s_{i_1}, \ldots, s_{i_k}$ then $L(M) = L(M, s_0, s_{i_1}) \cup \ldots \cup L(M, s_0, s_{i_k})$ so it suffice to prove the theorem for machines with a single start and single accept state, that is for languages of the form $L(M, s_0, s_k)$. We do this by induction on the number of edges of the partial deterministic automaton.

Consider the regular language $L(M, s_0, s_k)$. If there are no edges in $M$, then $L(M, s_0, s_k) = \{\epsilon\}$ or $L(M, s_0, s_k) = \emptyset$ as required. Suppose that $M$ has at least one edge, say with label $a$ from state $s_i$ to state $s_j$. Let $M_0$ be the partial deterministic automaton obtained by removing this edge. Then

$$L(M, s_0, s_k) = L(M_0, s_0, s_k) \cup L(M_0, s_0, s_i)(aL(M_0, s_i, s_j))^*aL(M_0, s_j, s_k).$$

Since $M_0$ has fewer edges the various $L(M_0, s_p, s_q)$ are empty or belong to $\Theta$. Since $L(M_0, s_0, s_k)$ is obtained from them by concatenation, union and starring it follows that $L(M, s_0, s_k) \in \Theta$ or $L(M, s_0, s_k)$ is empty. So by induction the theorem follows. \hfill \Box

A consequence of this result is that any regular language can be described by a so called regular expression. (In fact the proof tells us how to construct such a regular expression.) For example, $\{a^nb^m \mid n, m \geq 0\}$ can be described as $\{a\}^*\{b\}^*$ while the related language $\{a^nb^m \mid n, m \geq 1\}$ can be described as $\{a\}\{a\}^*\{b\}\{b\}^*$. A slightly more complicated example is the language consisting of all words containing two successive $a$’s or two successive $b$’s which can be described as $\{a, b\}^*\{aa, bb\}\{a, b\}^*$. 
Of course here \( \{a, b\} = \{a\} \cup \{b\} \) is a union, \( \{aa\} = \{a\}\{a\} \) is a concatenation, and so on.

The notation we have used here is not standard. There are various notations in use for regular expressions in conjunction with different operating systems, word processors and text searching programs. Complementation is often included in addition to the above three operations.

**Some algorithms concerning automata:** We think of a regular language \( L \) as being described by a (deterministic) finite state automaton \( M \) having \( L = L(M) \) as its language. Suppose that we are just given \( M \) as a list of symbols including a table describing \( \tau \) as above, and that we don’t know anything else about \( L = L(M) \). Some obvious questions are: Is \( L \) empty? Is \( L \) finite? If so, then what are the words in \( L \)?

All of these questions can be answered effectively. Based on estimates concerning the lengths of accepted words, one can give solutions which are effective in principle, but not very efficient. We also sketch much more efficient solutions for two of these questions.

First we consider determining whether \( L(M) \) is non-empty. Let \( n \) be the number of states of \( M \). Suppose that there is some word \( z \in L \) having length \( k \geq n \). Now the graph \( \Gamma_M \) has \( n \) vertices and \( z \) corresponds to a path from the start state \( s_0 \) which traverses \( k \geq n \) edges and ends in an accept state \( s(k) \). Since these \( k \) edges have \( k + 1 \) endpoints, some state must be visited twice. Thus \( z = uvw \) where \( v \) corresponds to a non-trivial loop in \( \Gamma_M \) at some state \( s(j) \). But then we may omit \( v \) to obtain a shorter word \( z_1 = uw \) corresponding to a path in \( \Gamma_M \) also ending at \( s(j) \). Thus \( z_1 \in L \). Continuing in this way we eventually find that \( L \) contains a word of length less than \( n \). Hence if \( L \) is non-empty, then \( L \) contains a word of length less than \( n \). This proves the following:

**Theorem 11.** If \( M \) is a finite state automaton having \( n \) states, then \( L(M) \) is non-empty if and only if \( M \) accepts some word of length less than \( n \). Hence, we can effectively determine whether or not \( L(M) \) is empty.

The obvious “in principle” algorithm is to simply check to see whether \( M \) accepts any of the finite set of words of length less than \( n \). If \( m \) is the number of symbols in \( \Sigma \), there are \( \frac{m^n - 1}{m-1} \) such words which can be a rather large number.

We now describe a more efficient method. Start at \( s_0 \) in \( \Gamma_M \). If \( s_0 \) is an accept state, we are done because \( M \) accepts the empty word. Call a state **accessible** if it can be reached from \( s_0 \) by a path along oriented edges. We want to know whether any accept state is accessible. Mark \( s_0 \) as being accessible. For each of the \( m \) edges leaving \( s_0 \) mark their destinations as being accessible. Now proceed inductively. For
each edge marked accessible at the last step and each edge leaving them, mark their destinations as accessible. Continue until either an accept state gets marked as accessible (and then $L$ is non-empty) or all destination edges from our marked states are already marked and none are accept states (so $L$ is then empty). This completes the algorithm. Note that the number of steps is less than $nm$ which is considerably quicker than looking at all words of length less than $n$.

We have performed a breadth first search of the graph of the FSA. Suppose now that we want to know whether or not $L = L(M)$ is finite. By the pumping lemma $L$ is infinite if and only if it contains some word of length at least $n$. Suppose that there is some word $z \in L$ having length $k \geq n$. Then as we have seen, the path corresponding to $z$ in $\Gamma_M$ contains a loop. If $z$ contains more than one loop, choose a subword $v$ corresponding to the shortest such loop and remove it to obtain $z_1$ as above. Continue in this way until we obtain a word $z_k \in L$ which has precisely one loop, so $z_k = u_kvkw_k$ where $v_k$ is the only non-trivial loop in $z_k$. Now $u_k$ and $w_k$ together have fewer than $n$ symbols since $u_kw_k \in L$ has no loops. Also $v_k$ is a simple loop as it has no subloops and hence it can visit each state at most once. Thus $v_k$ has length at most $n - 1$. Thus the length of $z_k$ is less than $2n$. But all of the words $u_kv_i^iv_kw_k$ for $i \geq 0$ belong to $L$ and at least one of them must have length $\lambda$ where $n \leq \lambda \leq 2n$ because of our estimates. Hence we can conclude the following.

**Theorem 12.** If $M$ is a finite state automaton having $n$ states, then $L(M)$ is infinite if and only if $M$ accepts some word of length $\lambda$ where $n \leq \lambda \leq 2n$. Hence we can effectively determine whether or not $L(M)$ is infinite.

Again the obvious “in principle” algorithm is to simply check to see whether $M$ accepts any of the finite set of word of length between $n$ and $2n$ inclusive. This is an even larger number of words.

We describe a more efficient method for this problem as well. In this algorithm we need to keep track of the states visited en route to an accessible state, which we think of as the ancestry of that state. A state will lie on a loop exactly when it is an ancestor of itself. We can keep track of ancestry for instance by forming a table with $n$ rows and $n$ columns labeled by the states $s_0, \ldots, s_{n-1}$.

Start at $s_0$ in $\Gamma_M$. Again call a state accessible if it can be reached from $s_0$ by a path along oriented edges. We want to know whether any accept state is accessible by a path containing a non-trivial loop.

Do not mark $s_0$ as being accessible yet. For each of the $m$ edges leaving $s_0$ mark their destinations $s_i$ as being accessible from $s_0$ by placing a mark in row $s_i$ and column $s_0$. Now proceed inductively. For each edge $s_i$ marked accessible at the last step and each edge leaving them, mark their destinations $s_i$ as accessible by placing marks in row...
$s_i$ in the following columns: (1) in column $s_j$ and (2) in all the columns which already had marks in row $s_j$. (Here $s_i$ is an ancestor of $s_j$, so we are forcing $s_j$ as well as the ancestors of $s_j$ to be ancestors of $s_i$.) Continue until all destination states are already marked with all the ancestral markers of their departure states.

There is a non-trivial loop in $\Gamma_M$ at state $s_i$ if and only if $s_i$ is marked as an ancestor of itself. This is easily determined by looking in the ancestry table. Now examine the accept states. For each accept state $s_k$, check to see if any of its ancestors has a non-trivial loop. $L$ is infinite if an only if one of these checks gives a positive answer. This completes the algorithm.

A crude bound on the number of steps required for this algorithm is $(nm)^2 + n^2$ which is considerably quicker than looking at all words with length in the given range.

As a consequence of the above, if we know that $L(M)$ is finite, then we can effectively find all of the words of $L(M)$, for they have length less than the number of states of $M$.

**Corollary 13.** Suppose $M$ is a finite state automaton having $n$ states and that $L(M)$ is finite. Then all of the words in $L(M)$ have length less than $n$. Hence, we can effectively find all the words in $L(M)$. 
1.1. The following is the graph of a partial deterministic automaton $M$.

Use this machine to determine which of the following three words belong to $L(M)$: $aaba$, $baba$, $ababb$. By adding another state and suitable edges, convert the given graph to the graph of a finite state automaton with the same language. Describe $L(M)$. Determine the cones of $L(M)$.

1.2. Which of the following strings are accepted by the machine described below:

(a) $aba$; (b) $abb$; (c) $a^{20}ba$; (d) $a^{20}bab$;
(e) $aabbbaa$; (f) $aabbaaba$; (g) $b$; (h) $bb$;
(i) $bbaa$; (j) $bba^{20}b$; (k) $baaaba$; (l) $baaaba$.

1.3. Describe the language accepted by the following finite state automaton:

(a)
1.4. Construct a (fully determined) FSA on the language $\Sigma = \{a, b, c\}$ that has start state $s_0$ (reject) and accept state $s_1$ and transition function

\[
\begin{align*}
\tau(s_0, a) &= s_0, \\
\tau(s_0, b) &= s_1, \\
\tau(s_0, c) &= s_1, \\
\tau(s_1, a) &= s_0, \\
\tau(s_1, b) &= s_1, \\
\tau(s_1, c) &= s_0.
\end{align*}
\]

1.5. Write down the transition function for the finite state automaton found in Exercise 1.
1.6. Which of the following strings are accepted by the non-deterministic finite state automaton below:

(a) abbba; (b) babb; (c) baabba; (d) a;
(e) b; (f) ab; (g) bb; (h) babab.

Describe the language accepted by this machine.

1.7. Let $\Sigma = \{a, b\}$ and let $L$ be the set of all words which contain two consecutive occurrences of $a$. Determine all of the cones $C_w(L)$. Draw the graph of a (deterministic) finite state automaton whose language is $L$.

1.8. Let $\Sigma = \{a, b\}$ and $L = \{a, bb, ab, abb\}$. Determine the cones $C_w(L)$ of this language and construct a machine that accepts this language.

1.9. Let $\Sigma = \{a, b\}$ and let $L$ be the set of all words which have length a multiple of 3. Determine all of the cones $C_w(L)$. Draw the graph of a (deterministic) finite state automaton whose language is $L$.

1.10. Let $\Sigma = \{a, b, c\}$ and let $L$ be the set of all words which contain the (consecutive) subword $abc$. Determine all of the cones $C_w(L)$. Draw the graph of a (deterministic) finite state automaton whose language is $L$.

1.11. Let $\Sigma = \{a, b\}$ and let $L = \{aab, abb\}$. First draw the graph of a partial deterministic automaton whose language is $L$. Then, adding a fail state if necessary, expand this to the graph of a (deterministic) finite state automaton whose language is $L$. Determine all of the cones $C_w(L)$. Does the finite state automaton you have found have the minimum number of states? Explain your answer.

1.12. Show that any finite language is regular. (Suggestion: consider the number of cones the language can have.)

1.13. Let $\Sigma = \{a, b, (, )\}$ and let $L$ be the set of all words with balanced parentheses, that is (1) containing the same number of (’s as )’s and (2) any initial segment has at least as many (’s as )’s. Show that $L$ is not a regular language. Find a regular sub-language $L_0$ of $L$ containing words having an arbitrarily large equal number of left and right parentheses.