Acknowledgement

Today's lecture note is entirely from Professor Reingold's lecture note for CS330 Spring 2015.

Contents

1 Finite state machines 2
  1.1 Some examples of their use .................................................. 2
  1.2 Finite state machines and juggling ......................................... 4
  1.3 Finite state machines as mathematical objects ........................... 4
  1.4 Finite state machines as a computational model ....................... 5

2 Closure properties of regular languages 7

3 Non-determinism 9
  3.1 An example of non-regular language ....................................... 10

4 Regular Expressions 10

5 Languages that are not regular 11

6 Graphs and finite state machines 11

7 An application: string matching 12
1 Finite state machines

1.1 Some examples of their use

First of all, let’s take a look at a state diagram that a typical CS 330 student might follow.

One simple problem that will serve to introduce the notion of state is that of counting words: given an input string, scan the string from beginning to end and count the number of words in the string. Simply counting the number of spaces (and other word separators, such as punctuation marks) will not do, as words may be separated by more than one such separators. Rather, we need to distinguish between two states:

From each of these two states we can define appropriate transitions: that is, if we are in a state and we see an input character, what state do we emerge in? If we are in the middle of a word and we see a letter, we remain in the middle of a word. If we are between words and we see a letter, we are now in the middle of a word. Likewise, if we are in the middle of a word and we see punctuation, we are now between words, and if we are already between words and we see punctuation, we are still between words.
Where do we start? That is, before we begin processing material, which state should we be in? We should start in the “between words” state.

Finally, since we would like to count words and not just flutter between two states, when do we increment our counter? We’d like to claim that we’ve counted another word when we follow the transition from “between words” to “in the middle if a word.”

This leads to the following state transition diagram:

Let us consider a slightly more complicated problem. We are given a sorted list of page numbers and would like to generate an appropriate index listing — that is, runs of three or more consecutive page numbers should be collapsed. For instance, given the input 5, 6, 7, 8, 11, 12, 14, 17, 18, 19, 21, 27, 28, we would like to generate the listing 5–8, 11, 12, 14, 17–19, 21, 27, 28.

This can be modeled with three states: NO, YES and MAYBE. We are in NO when we are not in the middle of a run, in YES when we are in the middle of a run, and in MAYBE when we may be in the middle of a run.

Appropriate transitions yield the following state diagram:

Let’s consider another example. A man with a wolf, goat, and cabbage is on the left bank of a river. There is a boat large enough to carry the man and only one of the other three. The man and his entourage wish to cross to the right bank, and the man can ferry each across, one at a time. However, if the man leaves the wolf and goat unattended on either shore, the wolf will surely eat the goat. Similarly, if the goat and cabbage are left unattended, the goat will eat the cabbage. Is it possible to cross the river without the goat or cabbage being eaten?¹

¹This problem appears in a medieval Latin manuscript by Alcuin of York. It was used in “Gone Maggie Gone,” a 2009
The problem is modeled by observing that the pertinent information is the occupants of each bank after a crossing. There are 16 subsets of the man ($M$), wolf ($W$), goat ($G$), and cabbage ($C$). A state corresponds to the subset that is on the left bank. States are labeled by hyphenated pairs such as $MG-WC$, where the symbols to the left of the hyphen denote the subset on the left bank; symbols to the right of the hyphen denote the subset on the right bank. Some of the 16 states, such as $GC-MW$, are fatal and may never entered by the system.

The “inputs” to the system are the actions the man takes. He may cross alone (input $m$), with the wolf (input $w$), the goat (input $g$), or cabbage (input $c$). The initial state is $MWGC-\emptyset$ and the final state is $\emptyset-MWGC$. Here is the transition diagram:

![Transition Diagram](image)

There are two equally short solutions to the problem, as can be seen by searching for paths from the initial state to the final state (which is doubly circled). There are infinitely many different solutions to the problem, all but two involving useless cycles.

### 1.2 Finite state machines and juggling


### 1.3 Finite state machines as mathematical objects

We need five entities to describe a finite state machine:

episode of *The Simpsons* in which Homer is trapped on one side of a river with his baby Maggie, his dog, and a bottle of poison capsules. He has only a flimsy boat so he can carry only one item at a time. If he takes the dog, Maggie might swallow some poison; if he takes the poison, the dog might bite Maggie.
• $\Sigma$ — a finite input alphabet
• $S$ — a set of states
• $s_0 \in S$ — an initial state
• $F \subseteq S$ — a set of accept states
• $\delta : S \times \Sigma \rightarrow S$ — a transition function

Then $M = (\Sigma, S, s_0, F, \delta)$ specifies a finite state machine.

We define a regular language to be a language recognized by some finite state machine.\(^2\)

### 1.4 Finite state machines as a computational model

As we have seen, finite state machines are a useful programming tool. They are also useful for exploring the notion of computation. A computer has a finite amount of memory, and thus can be in any of a finite (quite large, but still finite) number of states. As the computer receives input, it changes state.

Let us refine our notion of a finite state machine.\(^3\) A finite state machine receives input symbol-by-symbol; as each symbol comes in, it changes state according to the appropriate transition. As well, certain states are declared as accept or final states. If, at the end of an input string, the machine is in an accept state, it is said to accept the input; otherwise it rejects the input. Thus a finite state machine recognizes a language, some subset of all finite strings over an alphabet.

Let us design a finite state machine to recognize even numbers in binary, where the input has the most significant digit first. As the machine processes the input string, it fluctuates between two states, one if the part of the input string it has seen so far is even, and the other if it is odd. The transitions are not hard to find, as a number in binary is even if it ends in a 0 and odd if it ends in a 1. Finally, we want this machine to accept if the input is even. Thus we have:

This technique can be expanded to slightly more interesting problems, for instance recognizing multiples of five (still in binary). Consider a multiple of five. If a 0 is appended to the end, the number is doubled and is still a multiple of five. If a 1 is appended to the end, the number is doubled and 1 is added, so we should move to state 1. Similar reasoning for the other four states gives us:

\(^2\)Rosen follows a different treatment by defining regular languages differently and then deriving this as a theorem. This is not the approach we will be following in lecture. You have been warned.

\(^3\)Finite state machines are also known as finite automata. In particular, the variety we are developing here is a deterministic (or definite) finite automaton.
What about multiples of ten? We can follow a similar approach and find the transitions for ten states labeled 0–9. Alternatively, since the multiples of ten are the numbers that are both even and multiples of five, and we have machines that recognize even numbers and multiples of five, it would be nice if we could somehow combine these two machines. In essence, we want to run these two machines in parallel, applying the input to both and accepting the input if both machines accept.

We can collapse these two machines into one by keeping track of the state of each machine in our new machine. This gives us the (somewhat messier) machine.\footnote{In fact, in this case, this is the identical machine, with the states relabeled, that we would derive if we used the same approach we did for the multiples of five on this problem. There is a simple way to reduce this machine to one with only six states: do you see how to do it?}
2 Closure properties of regular languages

We can prove that a language \( L \) is regular by constructing a finite state machine that recognizes \( L \). We use this observation to prove a number of results.

**Lemma:** The set of regular languages is closed under intersection.\(^5\)

**Proof:** Let \( L_1 \) and \( L_2 \) be regular languages. Say that \( L_1 \) is recognized by \( M_1 = (\Sigma, S, s_0, F, \delta) \) and \( L_2 \) is recognized by \( M_2 = (\Sigma, T, t_0, G, \lambda) \). Then \( L_1 \cap L_2 \) is recognized by \( M = (\Sigma, S \times T, (s_0, t_0), F \times G, \alpha) \), where \( \alpha: \Sigma \times S \times T \to S \times T \) is defined as \( \alpha(x, s, t) = (\delta(x, s), \lambda(x, t)) \).

**Lemma:** The set of regular languages is closed under union.

**Proof:** Let \( L_1 \) and \( L_2 \) be regular languages. Say that \( L_1 \) is recognized by \( M_1 = (\Sigma, S, s_0, F, \delta) \) and \( L_2 \) is recognized by \( M_2 = (\Sigma, T, t_0, G, \lambda) \). Then \( L_1 \cup L_2 \) is recognized by \( M = (\Sigma, S \times T, (s_0, t_0), F \times T) \cup (S \times G, \alpha) \), where \( \alpha: \Sigma \times S \times T \to S \times T \) is defined as \( \alpha(x, s, t) = (\delta(x, s), \lambda(x, t)) \).

**Lemma:** The set of regular languages is closed under complementation.

**Proof:** Let \( L \) be a regular language. Say that \( L \) is recognized by \( M = (\Sigma, S, s_0, F, \delta) \). Then \( \overline{L} \) is recognized by \( M' = (\Sigma, S, s_0, S - F, \delta) \).

These three lemmas directly lead to the following closure theorem:

**Theorem:** The set of regular languages is closed under intersection, union, and complement.

We move to other properties of regular languages.

**Theorem:** All finite sets are regular.

**Proof sketch:** Consider the language \( L = \{ \sigma_1, \sigma_2, \ldots, \sigma_k \} \). We prove this by induction on \( k \).

Base case: Here \( k = 1 \). Assume \( \sigma_1 \) is not empty;\(^7\) let \( \sigma_1 = a_1 a_2 \cdots a_n \). We can construct a finite state machine that recognizes \( \sigma_1 \). \( \Sigma \) is the appropriate alphabet. Let \( S = \{ 0, 1, 2, \ldots, n, \text{FAIL} \} \), \( s_0 = 0 \), and \( F = \{ n \} \).

Define the transition function \( \delta \) as follows:

\[
\delta(\text{FAIL}, a_j) = \text{FAIL} \\
\delta(i, a_j) = \begin{cases} j & \text{if } i + 1 = j \\ \text{FAIL} & \text{otherwise} \end{cases}
\]

Then \( M = (\Sigma, S, s_0, F, \delta) \) recognizes \( \sigma_1 \) and no other strings.

Inductive step: For \( k > 1 \), build a machine \( M_k \) that recognizes \( \sigma_k \). By induction there is a machine \( M_{1..k-1} \) that recognizes the set \( \{ \sigma_1, \sigma_2, \ldots, \sigma_{k-1} \} \). By the “union lemma” above, the union of \( \{ \sigma_k \} \) and \( \{ \sigma_1, \sigma_2, \ldots, \sigma_{k-1} \} \) is therefore regular.

We now consider some slightly more complicated questions of regularity. Is concatenation closed over regular languages? The concatenation of languages \( L_1 \) and \( L_2 \) is defined as

\[
L_3 = L_1 L_2 = \{ l_1 l_2 | l_1 \in L_1, \ l_2 \in L_2 \}
\]

For example, if \( L_1 = \{ \text{all strings with at least two zeroes} \} \) and \( L_2 = \{ \text{all strings with at least two ones} \} \) then \( L_3 = L_1 L_2 = \{ \text{all strings that can be divided into a first section with at least two zeroes and a last section with at least two ones} \} \).

---

\(^5\)That is, the intersection of two regular languages is still a regular language.

\(^6\)This is precisely what we did to solve the problem of multiples of ten.

\(^7\)Exercise: Design a finite state machine that recognizes only the empty string.
So far we have connected machines in “parallel”, meaning that we simulate running two machines at the same time. For concatenation we want to hook up machines in “series”. Thus, suppose we have 2 arbitrary machines $M_1$ and $M_2$ accepting $L_1$ and $L_2$:

One attempt at making a machine $M_3$ to accept $L_3$ involves linking the final states of $M_1$ to the start state of $M_2$:

However, we have some problems here. What is the boundary between the two machines? If we are given a string in $L_3$, how do we decide which part of it should be checked for membership in $L_1$ and which part should be checked for membership in $L_2$. Moreover, the transition from machine $M_1$ to $M_2$ does not use up an input character, which violates the definition of a finite state machine.

We can fix the second problem by connecting any transitions into an accepting states of $M_1$ directly to the start state of the second machine as follows:

The first problem with our construction, however, is more complicated. In some sense, we want to break up a string in all possible ways $x = uv$, checking if $u \in L_1$ and $v \in L_2$. The basis for such a technique is nondeterminism.

Now we have

**Theorem:** If $L_1$ and $L_2$ are regular, then $L_1L_2$ is regular.
We define $L^*$ to be $\{\varepsilon\} \cup L \cup LL \cup LLL \cup \cdots$. With the above theorem and the fact that regular language is closed under union operation, it would wrong to conclude that if $L$ is regular, then $L^*$ is regular because that is an infinite union and the union closure property holds only for finite unions. However $L^*$ is regular if $L$ is; the next section shows us how to approach the problem.

3 Non-determinism

*When you come to a fork in the road, take it.*

—Yogi Berra

A non-deterministic finite state machine transitions from one state to a whole set of states. In other words, it tries out several possibilities at the same time, and accepts if any of the possibilities lead to an accepting state.

More precisely, a non-deterministic finite state machine is a quintuple $(\Sigma, S, s_0, F, \delta)$ where:

- $\Sigma$ is the alphabet of the machine
- $S$ is the finite set of states of the machine
- $s_0$ is the start state
- $F$ is the set of final states of the machine
- $\delta$ is a mapping from $S \times \Sigma \rightarrow 2^S$. Where $2^S$ is the set of all subsets of $S$; it is sometimes called the power set of $S$ and denoted $P(S)$. For example:

$$2^{\{1,2,3\}} = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{2,3\}, \{1,3\}, \{1,2,3\}\}$$

Thus, this transition function maps states and input characters to a set of states.

A non-deterministic finite state machine accepts if any of the states in which it ends up at the end of the input is an accepting state.

In a non-deterministic machine, we may also allow transitions from state to state without gobbling up input; these transitions are called epsilon transitions ($\varepsilon$ is the null string). Neither nondeterminism nor epsilon transitions really increase the power of our finite state machines, because they can all be modeled with a regular deterministic finite state machine. Nevertheless, they do make life easier. For example, we may now represent the concatenation of two finite state machines as the following nondeterministic machine:

![Diagram of non-deterministic finite state machine](image)
Using nondeterminism, we can also show that for any given a regular language $L$, the $L^{\text{reverse}}$ is also regular. $L^{\text{reverse}}$ is simple the language consisting of the reverses of all the strings in $L$. For example, the reverse of the string “I love CS330” is “033SC evol I”.

To prove closure under reversing, we start with the machine $M$ that accepts $L$. Then we reverse the direction of all the transitions, and switch starting states with final states. If $M$ has more than one final state, then we add a new starting state to the new machine, and add epsilon transitions from this new state to each of the final states of $M$.

Note that reversing the directions of transitions might mean that we have two transitions on the same symbol coming out of a state, so that the new machine needs to be nondeterministic.

Now, as an exercise, show that $L^*$ is regular when $L$ is by connecting final states to the start state with an epsilon move.

### 3.1 An example of non-regular language

Consider the language $L = \{0^n1^n | n = 0, 1, 2, \ldots \}$. This is not a regular language. If it was, we should have an finite state automaton which accepts $L$. Suppose this automaton has $N$ states, then if we input a string $0^i1^i$ of length $2i, i > N$, by the pigeon hole principle, when we move around and reach the first “1”, we move $i$ times and must hit a certain state at least twice. So there will be a loop starting and ending at this state. Cutting the substring which labels the loop or repeating this substring for any times will give us another string which is also accepted by the automaton. But the string can’t satisfy that we have the same number of “0”s and “1”s!

You can find a more detailed proof in Rosen. The same analysis gives us the pumping lemma: Let $L$ be a regular set. Then there is a constant $n$ such that if $z$ is any word in $L$, and $|z| \geq n$, we may write $z = uvw$ in such a way that $|uv| \leq n, |v| \geq 1$, and for all $i \geq 0$, $uv^iw$ is in $L$.

Pumping lemma is a very useful tool to prove that a language is not regular.

* Use the pumping lemma to prove that the set $\{1^n2^n | n = 0, 1, 2, \ldots \}$ is not a regular language. +10

### 4 Regular Expressions

Now we define a regular expression, which turns out to be equivalent to finite automata, but can be a nicer way to express some regular languages (those that are accepted by regular expressions and/or finite automata). The empty set (\(\emptyset\)) is a regular expression, corresponding to the language containing no words. Epsilon (\(\epsilon\)) is a regular expression corresponding to the language containing only the empty word. (Notice that there is a (somewhat subtle) difference between these two.) Also, each letter $x$ in our alphabet is a regular expression corresponding to the language containing that letter as its only word. We also define regular expressions to be closed under union, concatenation, and Kleene star. Using these properties together, it is possible to define quite complicated languages.

Since we have shown that regular languages have the same closure properties, it is perhaps not surprising that our “new” class of languages turns out to be exactly equivalent to the old one:

**Theorem:** Every regular expression represents a regular language, and every regular language can be represented by a regular expression.

**Proof:** By the rules of construction of regular expressions, the first part of this theorem is clear.

To prove the second part of the theorem, we use a variant on the Floyd-Warshall algorithm\(^8\) to construct a

---

\(^8\)The Floyd-Warshall algorithm, presented in section 25.2 of CLRS, is a $\Theta(|V|^3)$ dynamic-programming algorithm for solving
The regular expression given an arbitrary finite state machine.

Note that this algorithm assumes that the machine has exactly one final state. If it has multiple final states, the algorithm can be run once for each and the results joined with the + (union) operator on regular expressions.

Say the machine has \( n \) states. Give each state a unique integer between 1 and \( n \), where state 1 is the initial state and state \( n \) is the final state. Define the expression \( R^k_{ij} \) to be the regular expression describing strings that carry the FSM from state \( i \) to state \( j \), using only states between 1 and \( k \) as intermediate states. Then the expression that corresponds to the machine as a whole is \( R^0_{1n} \).

\( R^k_{ij} \) is defined recursively:

\[
R^0_{ij} = \{ x \in \Sigma | \delta(i, x) = j \} \\
R^k_{ij} = R^{k-1}_{ij} + R^{k-1}_{ij} (R^{k-1}_{kk})^* R^{k-1}_{kj}
\]

As a regular expression can be constructed given any FSM, any regular language can be expressed by a regular expression.

### 5 Languages that are not regular

We talked about pumping lemma which can help determine that a language is not regular. A classic example of these languages is \( \{0^n1^n | n = 0, 1, 2, \cdots \} \). There is not a finite state machine to recognize this language since you need infinite many states to keep track of “Where am I” during the computation. This requires infinite memory. For languages like the one above, we can recognize them using a finite state machine with a stack (PDA: push down automata): the machine just pushes every 0 into the stack until it sees the first 1. From that point it pops a 0 everytime it sees a 1. If it sees a 0 again, the string is rejected. Otherwise, if the stack is empty when the input ends, the machine accepts the input string. If a language can be recognized by a PDA, we call it context free language. (The name “context free” comes from the property of grammers generating this kind of language).

Can PDA recognize every language? No. The language \( \{0^n1^n2^n | n = 0, 1, 2, \cdots \} \) can not be recognized by any PDA. With a slightly advanced version of pumping lemma we can prove that this language is not context free.

How about a PDA with two stacks? It can recognize the above language. Actually a PDA with two stacks is equivalent to a Turing machine, which can recognize any language that is decidable and compute any problem that is computable.

Are there any language with is not decidable (that means, given the input string, no machine can tell you whether the input belongs to that language)? Yes. One example is that we can’t have a machine which, given input strings encoding a student’s program, test input and standard output, can tell you whether the students program generates the correct output, because there might be dead loops in the student’s code and the machine never “dares” to cut off the running!

### 6 Graphs and finite state machines

There are many questions about FSMs and regular languages that can be solved by treating the FSM as a graph. For instance, given an FSM, does it accept anything? Is a language infinite? Are two languages the all-pairs shortest-paths problem on a directed graph. Here we apply it to an FSM, treated as a graph.

9In terms of graphs, then, this question would be whether there are any paths from the start state to any of the accept state.
7  An application: string matching

One useful application of FSMs is in string matching.\(^\text{10}\) The basic idea is to construct an FSM that recognizes the string being searched for and to run it with the text as input.

Of course, this brute force approach is inefficient. In particular, as stated, it will only find the string if it occurs at the very beginning of the text. If the FSM rejects, we could then run it on the text beginning at the second character, and then beginning at the third character, and so on, but that would require \(O(mn)\) time, and we can do better. For instance, if the search string is \texttt{ababaca} and the text is \texttt{abababacaba}, after reaching the third \texttt{b} in the text, rather than announcing failure and giving up, we can make use of the fact that the text, so far, still matches the beginning of the search string, as long as we begin looking two characters into the text.

How can this idea be encoded in an FSM? The idea is simple: \textit{on failure, move to the latest state in the FSM that still matches some prefix of the text}. The algorithm is given in detail in CLR, and yields an \(O(m + n)\) string matching algorithm. This algorithm and its variants are commonly used in text editors.

\(^{10}\)This is discussed in Cormen, Leiserson, Rivest, and Stein, chapter 32