Languages (Finite State Machines)

Carol Zander

Finite state machines with no output

Many processes, algorithms, and machines can be modeled using a finite state machine (FSM). These have a finite set of states (obviously), one of which is the starting state, and a set of which are final states. They also have an input alphabet and a set of transition rules for moving from one state to another based on the input symbol. Some FSMs have output, some don't. These notes only cover those without output.

Kleene's Theorem

Any language that can be defined by

1. regular expression
2. finite state machine

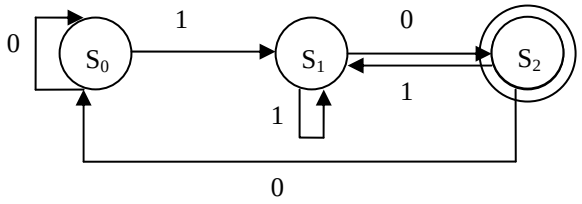
is a regular language. In other words, every regular expression (recognizing a regular language) has an equivalent FSM and vice versa.

When they have no output, FSMs are often called finite state automata (FSA). When they have a transition for every character in the input alphabet, they are called deterministic finite automata (DFA).

Formally, a finite state automaton consists of:

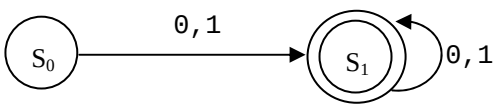
- S: Finite set of states
- A: Input alphabet
- t: Transition function that determines what the next state is given current state and input character
- S_0: Starting state ($S_0 \in S$)
- F: Final states ($F \subseteq S$)

A string is accepted (or recognized) if the FSA ends in one of its final states after processing all of the characters in the string. Here is an example of a finite state machine with no output. The final states are drawn as double circles. Other notation used is a minus (in the state circle) for the start state and a plus for a final state. In this example, the alphabet is {0, 1},

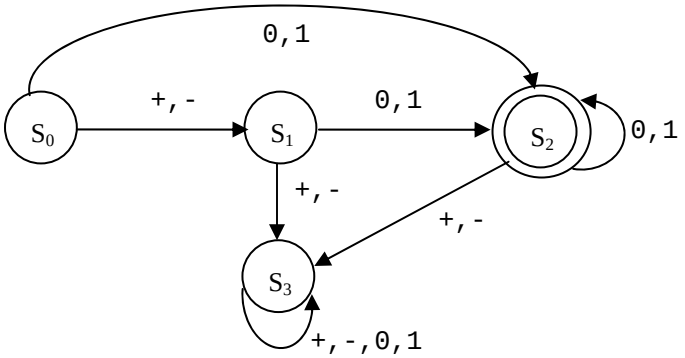


This is a DFA since every state has a zero or one labeled transition. What is the set of strings recognized by this DFA? A string that starts with any number of zeros keeps you in the start state. A one gets you to S_1. More ones keep us in S_1. A zero takes us to the final state S_2 but a one takes us back to S_1. Therefore, the only way to get to the final state is to get a one (to S_1) and then zero to S_2. This is the language of all binary numbers that end in "10." In other words, the language of all binary numbers divisible by two, but not divisible by four.

Some DFAs are straightforward, e.g., for unsigned binary numbers a one or zero get you to the final state, which allows more zeros and ones:



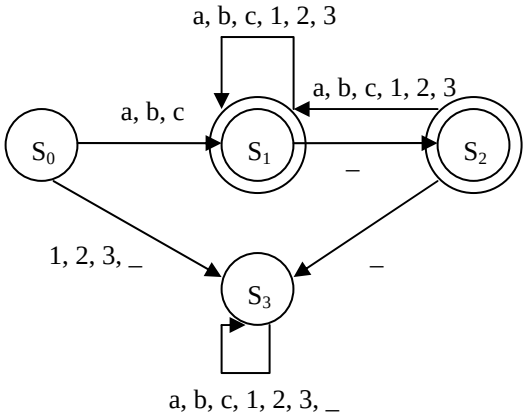
The DFA for signed binary numbers is larger, but still straightforward. Notice the state S_3 . It is called a trap state because once you get an illegal character (that causes the string to be known not to be in the language), you are trapped in that state. Note that if the empty string is in a language, then the start state will also be final.



Without state S_3, it is called a nondeterministic finite automaton (NFA), which means a solution exists to accept all strings in the language, but it may not be unique. If there is no transition for a character, then you immediately determine that the string is not in the language. To accept the string, determine the string is in the language, all characters must have been used in transitions and you must be in a final state. Differences in NFA (from DFA):

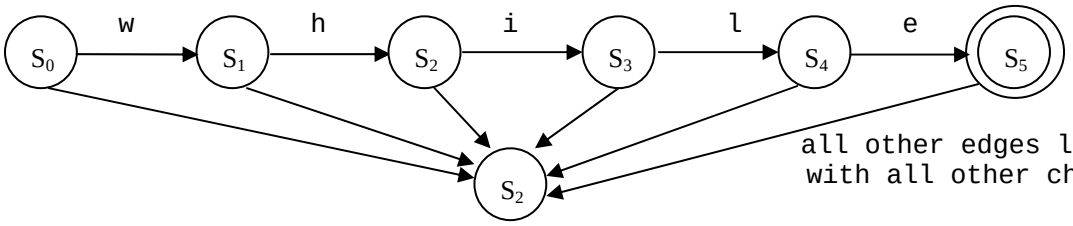
- There may not be edges for all characters out of every state
- There may be more than one edge out of a state labeled with the same character
- Edges can be labeled with a string (more than one character)
- Edges can be labeled with the empty string (change state without any input)

Here is a DFA for recognizing Ada-like identifiers using our reduced alphabet:

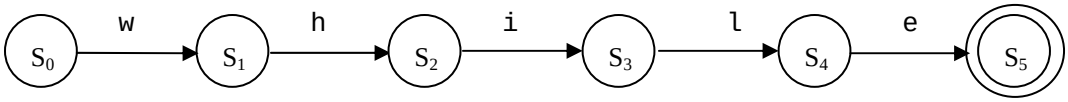


The S_1 state is where we get to after we've seen the initial letter and then have seen more letters and digits. We get to S_2 when we see an underscore. But if we see another underscore, we end up in the trap state. Note that not all finite automata have trap states. The example has two final states.

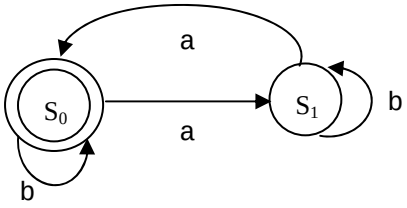
When you are drawing a DFA for a finite language such as "while", it is straightforward with all characters except the ones in the string taking you to the trap state:



The NFA is the same, but without the trap state:

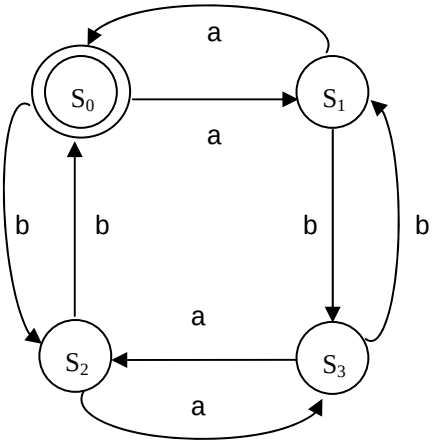


The DFA for the language of even a's, alphabet {a, b}, has states that could be called the *even* and *odd* states:

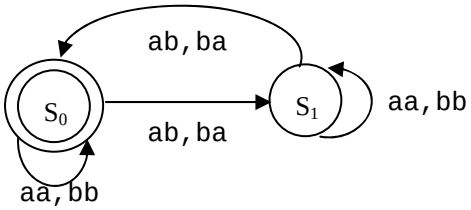


The DFA for even a's and even b's (recall the regular expression (aa | bb| (ab|ba)(aa|bb)*(ab|ba))*) expands this even/odd state idea. There are four states:

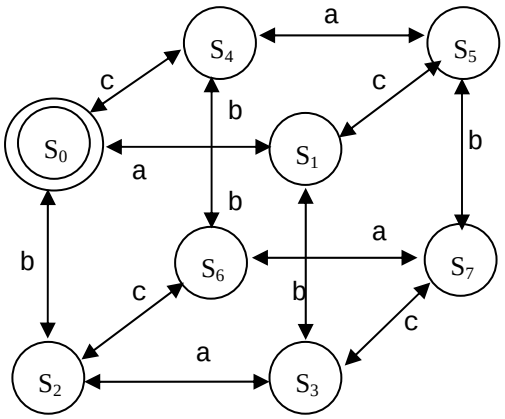
S_0 (start/final state)where a's and b's are even
S_1 where a's are odd and b's are even
S_2 where a's are even and b's are odd
S_3 where a's are odd and b's are odd



An NFA for the even a's, even b's language is drawn with only two states using strings on the transitions:

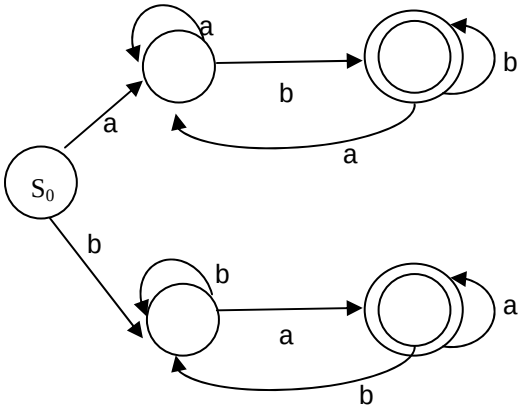


The DFA for the language even a's, even b's, even c's (alphabet = {a, b, c}) is straightforward with 8 states representing all the combinations of even/odd for 3 characters. If the edges are drawn with double arrows for simplicity to mean an edge in both directions, the DFA looks like a cube:



Often if I am trying to write a regular expression, and I am stuck, I will draw the finite automaton to help me write the regular expression. It doesn't always help because it is typically easier to draw the automaton than write the regular expression, but is often useful for grasping what is needed in the regular expression.

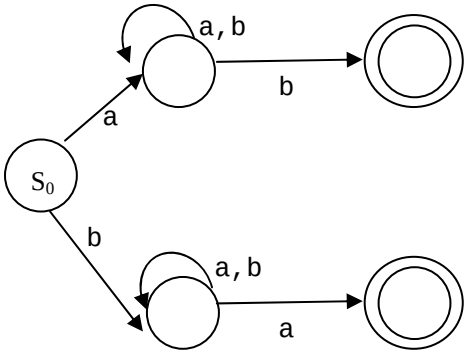
An example with alphabet {a, b}: all words that begin and end in a different letter. Often state labels are not used.



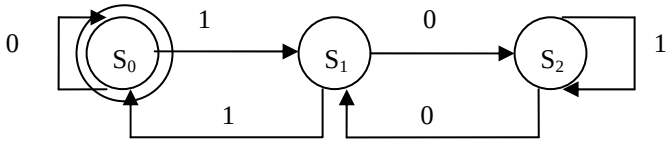
The regular expression for this language is easy to see from the DFA:

```
a(a|b)*b | b(a|b)*a
```

The NFA for the same language uses non-determinism in that there is more than one choice for a character transition out of a state. This demonstrates the existence of a way to a final state as opposed to determining it.



Another example: Can we construct a DFA to recognize binary numbers that are divisible by three? Yes. Since we examine the digits in order, we need to consider the effect of adding a zero or one at the end of a previous number. What we need to remember is what the remainder of the number is when divided by three. If we add a 0 to the number, we are (in effect) multiplying the number by two. If we add a 1 to the number, we are multiplying by two and adding one. We'll have three states (one for each remainder). Taking examples helps to see the nature of the states: 1=0001 2=0010 3=0011 4=0100 5=0101 6=0110 7=0111 8=1000 9=1001



Recall that (according to Kleene's theorem) every language recognized by an FSA also has a corresponding regular expression. Often determining the FSA is easier than determining the regular expression. There is an algorithm to write a regular expression from an FSA though. Doing it informally, for this case, we can build one using some trial and error.

1st try: (0 | 11)* (This accounts for all of the transitions that leave us in S_0, possibly going to S_1 first.)
2nd try: (0 | 1(00)*1)* (Now we also go to S_2.)
3rd try: (0 | 1(01*0)*1)*