Finite state machines

DECEMBER 31, 2021 ~ BOB

This is the second article in a series about some classic computer science:

1. Regular expressions
2. Finite state machines
3. Comparing regular expressions and finite state machines

Finite state machines are a way of checking that a series of inputs is valid, potentially doing some actions while you're doing this checking. They're not something I use all that often, but if I spot that some logic is turning into a tangled mess of if / else it's sometimes a clue that I should rip it out and replace it with a finite state machine. I'll go into an example implementation below – as you'll see they're not tricky.

I'll concentrate on the simplest version for most of this article, whose full name is *Deterministic Finite State Machine*. Later in the article, I'll briefly touch on the Non-deterministic Finite State Machine, and also the Hidden Markov Model. Finite state machines are sometimes known as *finite state automata* or just *state machines*. For most of the article, I'll use the abbreviation *FSM* to mean *deterministic finite state machine*.

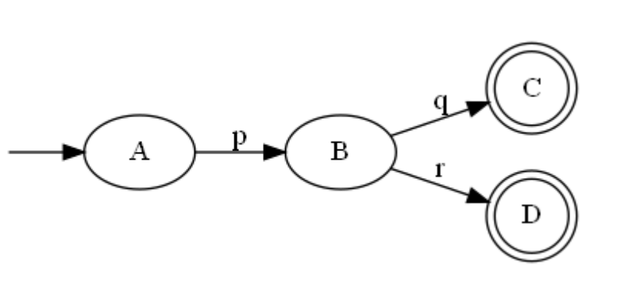
With all that out of the way: what is a FSM?

Basics

A FSM is a bit like a game of hopscotch that you build for your computer. In case you've never played it, hopscotch is where there are squares marked out on the ground, and the squares are laid out roughly in some kind of line. You start at one end of the line and try to get to the other end. As you move along the line, your foot can be in only one square at a time. (Sometimes you need to put both feet down on the ground, each in different squares, but most of the time you're hopping so only one foot is on the ground.)

Image credit

This is a FSM:



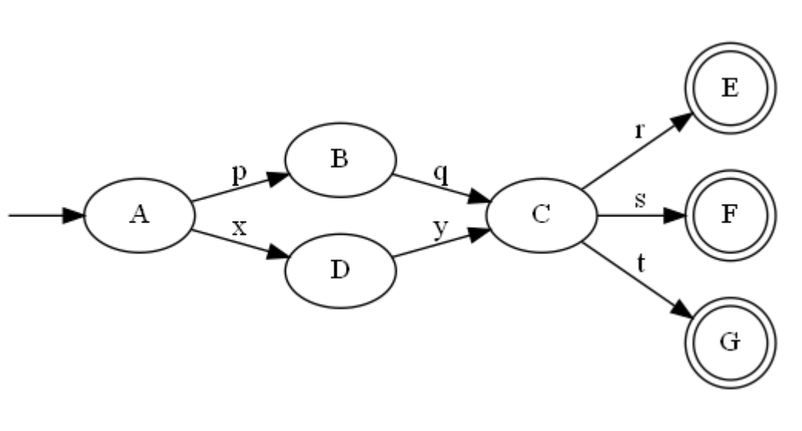
A, B, C, and D are states, which are like the squares in hopscotch. The arrows are the only ways you can move from one state to another, so you can't e.g. go directly from A to D. You start by following the arrow that has no blob at its start, which means you start in state A. The labels on the arrows are what input you must receive in order to follow that arrow, i.e. to go from A to B you must receive input p.

C and D are marked as special by having a double outline. These are *accepting states*. If the computer is in an accepting state when the input finishes, then the input is valid. If the computer is in a different state when the input finishes, the input is invalid.

So, the FSM above will accept *pq* and *pr* as valid inputs. The order matters – *qp* and *rp* are invalid.

Memory

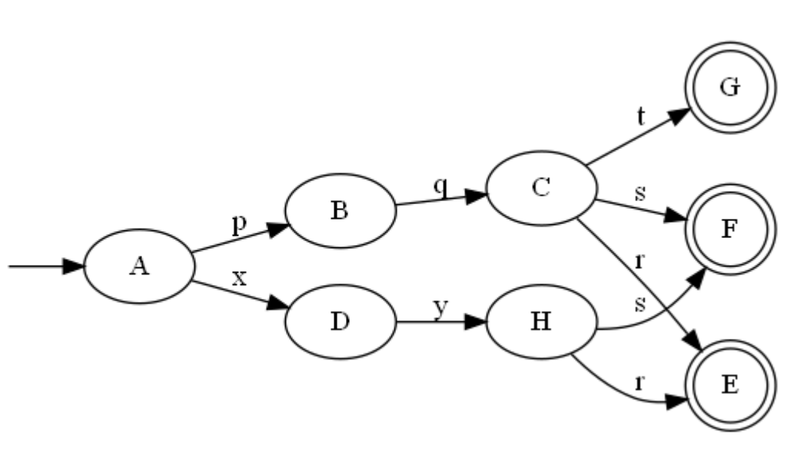
The FSM above is a relatively simple one. This is a slightly more complicated one:



Paths through the FSM can split or join as needed. This FSM will accept *pqr, pqs, pqt, xyr, xys* or *xyt*.

It's important to note that once the FSM is in state C it has **no** memory of which path it took to get there (i.e. via B or via D). The only thing it knows is where it is now and where it can get to next based on which input it gets.

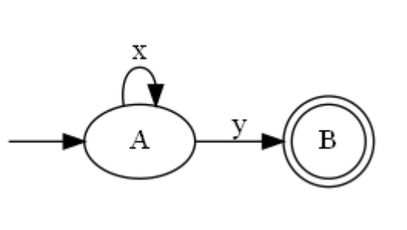
If you want the branches available after C to depend on which path you took to get to C then you need to split the FSM up by creating new states. For instance, if you wanted to restrict the input such that it can only end in t if it started with p, you could build a FSM like this:



There's now a new state H, which is a bit like C but with different paths leading from it.

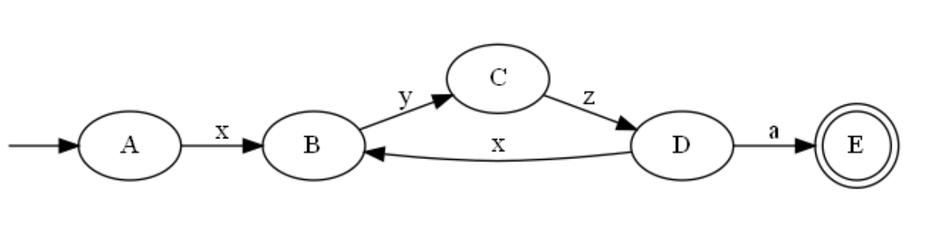
Loops

Loops are valid in a FSM. They can be self loops (looping from a state immediately back to itself):



This FSM will accept *y, xy, xxy, xxxy* and so on.

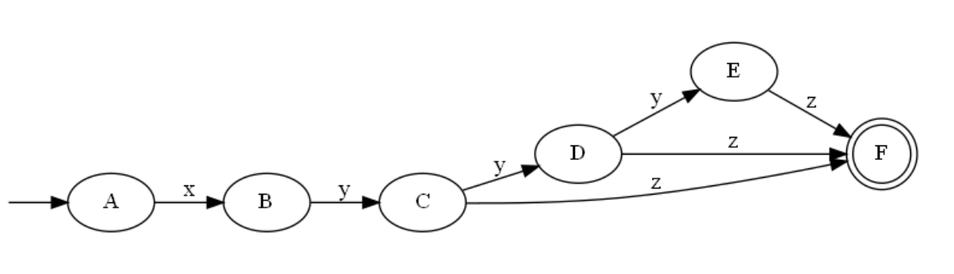
They can also be longer loops involving more states:



This will accept *xyza, xyzxyza, xyzxyzxyza* etc.

Note that, as before, the FSM has no memory of where it has been before. This includes how many times it has gone around a loop. If the reason why you'd like the loop to remember its past is that you want to limit the number of repetitions to e.g. 1 to 3, then unfortunately you need to do this by copying out the relevant bit of the FSM several times like this.

For instance, if you want to accept only *xyz, xyyz* or *xyyyz* you could do it with this FSM:



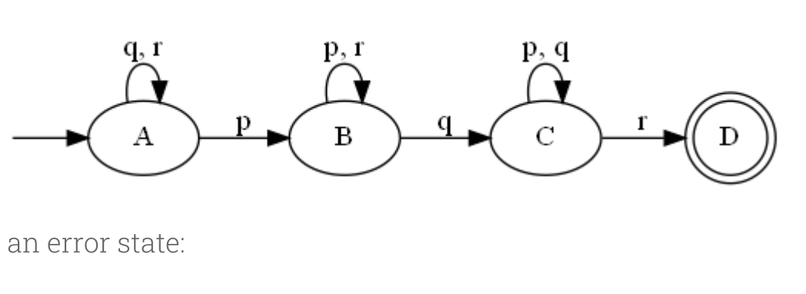
Errors

So far I've shown only happy paths and ignored errors. How errors should be handled depends on your context. The options are likely to be some variation on:

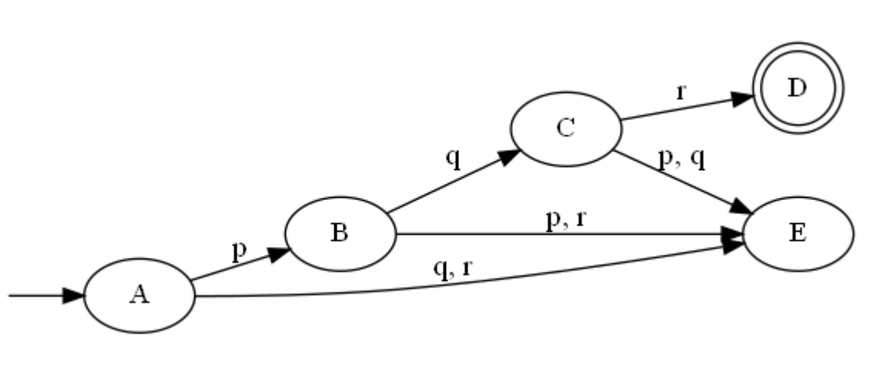
1. You ignore bad inputs
2. Bad input sends the computer to a bad state from which there's no escape
3. Bad input wipes out the progress through the FSM, by returning to the start

If the only possible inputs are p, q or r, and you want to accept only *pqr* (in that order) then you can model the three options above as follows.

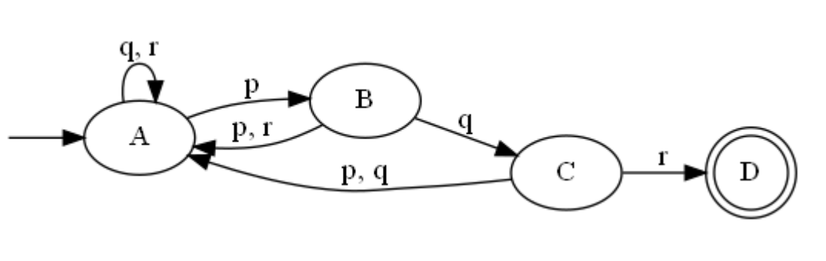
Ignoring bad inputs:



Going to an error state:



Going back to the start:



When and how you might use one

So far I've been presenting this in a fairly abstract way, using single letters for states and inputs. This was to present the general concepts in as simple a way as possible. In practice, it's helpful to label your states with longer and more descriptive text, that highlights what's important about that state. That could be something like *PayloadReceived*, *AttachmentAdded* etc. Similarly, it's unlikely that your inputs would be just a series of single letters. They could be things like messages taken from a queue, requests received by some code implementing a REST API etc.

So, you might use a FSM to support a micro-service, where it's got several stages of work to get through for one business transaction, but it can't do all the stages based on a single input message / API call. A FSM could keep track of how far a given business transaction has progressed, and what remains to be done.

If your logic is simple enough, then you can cope without using a FSM. Instead you would use a series of variables (probably booleans) to keep track of what's already happened, and if/else statements to implement the logic of how the next step can depend on what's already happened and what the current input is.

As the complexity and/or size grow too big this becomes unmanageable, and a FSM is often a good alternative. It could be that things continue to grow, and the FSM becomes unmanageable. In this case it might be possible to split the FSM into two or more separate FSMs. How this split works depends on context. One possibility is that FSM A becomes detail that's inside a single state in FSM B, i.e. there are two nested FSMs. Another possibility is that FSM A follows FSM B – they're at the same rank in a hierarchy, but one after the other.

Implementing

If you ignore the details of determining which input you've received from the set of all possible inputs, then implementing a FSM is relatively straightforward. You just need to store a list of states, and for each state the list of input / next state pairs. Unlike with regular expressions, I have always found it's simple enough to implement a FSM directly myself rather than using a library or toolkit.

I suggest that you *don't* identify the states by their position in the list or array, i.e. state 0, 1, 2 etc. This has two problems – the names aren't very descriptive, and it's hard to change the FSM in the future. (If you add an extra state between states 6 and 7, then all states 7 and upwards need to be renumbered, with all the references to them too.) You could give the states names that are text strings, as that is descriptive. However, you won't get any helpful from the compiler about spelling the names correctly when you want to use a state as a destination somewhere else.

So I suggest that you use something like an enumeration, where the members of the enumeration have descriptive names.

You then have to think about the transitions between states. Is it possible that one transition could be associated with several different inputs, or is each transition only associated with one input? If you could have many inputs per transition, do you want to represent it directly like that, or do you want to represent it as many different transitions between the same two states but with a different input per transition?

The final things are some way to indicate which states are accepting states – a boolean per state is probably enough for this – and some way to indicate the initial state.

You then loop through the inputs, and for each input looking up that input in the transitions out from the current state to see which state to go to next. At the end of the input you see if the current state is an accepting state or not.

One possible implementation in C# (with no error handling, and only simple string inputs) is as follows:

```
using System;
using System.Collections.Generic;
public class Program
{
    enum StateName
    {
        Init = 0,
        HeaderReceived = 1,
        BodyReceived = 2,
        FooterReceived = 3
    }

    class State
    {
        public bool isAccepting = false;
        public Dictionary<string, StateName> transitio
    }

    public static void Main()
    {
        Dictionary<StateName, State> fsm = new Diction
        {
            { StateName.Init, new State { transitions =
            { StateName.HeaderReceived, new State { tra
            { StateName.BodyReceived, new State { trans
            { StateName.FooterReceived, new State { isA
        };

        List<string> inputs = new List<string> { "h", 

        StateName currentState = StateName.Init;

        foreach(string input in inputs)
        {
            StateName nextState = fsm[currentState].tra
            currentState = nextState;
        }

        if (fsm[currentState].isAccepting)
        {
            Console.WriteLine("input is valid");
        }
        else
        {
            Console.WriteLine("input is invalid");
        }
    }
}
```

It's checking inputs to see if they contain a header, one or more body sections and then a footer.

Beyond just validation

So far I've talked about the smallest and simplest version of FSMs, where they give a yes/no answer to whether a series of inputs is valid. While this can be useful, sometimes you also need to do some processing based on the input. For instance, the inputs deliver you the data that you will build into some output object.

This is usually quite straightforward. You extend the definition of each state so that it can specify some code that needs to be executed when you enter that state. There might need to be some help such that the different lumps of code attached to the different states can communicate with each other to get some overall task done, e.g. by reading and writing an object that's created before the inputs are processed, and then passed to each lump of code as it's invoked.

In the example FSM above, it is processing something like an email. The init state could create an email builder object, and add it to the shared object. When the header is received, it can be passed to the AddHeader method of the builder. Similarly, when a body section or footer are received, they can be passed to AddBodySection and AddFooter. If the loop through the inputs ends with the code being in an accepting state, it can call GetEmail on the builder to get the email.

Non-deterministic FSMs

All the FSMs up till now have been *deterministic*. That means it's always possible for a given (state, input) pair to know which next state to move to. Depending on how you handle errors, there will be either exactly one next state, or zero or one next states (zero next states implies you must stay in your current state). You also don't need any other information beyond the current state and current input to know what to do next.

This seems like I'm labouring the point, but they are all assumptions that we've been making so far. A non-deterministic FSM invalidates (NFSM) those assumptions. With a NFSM, there can be two or more output states for a given (state, input) pair. How can you work out what to do next? You could either pick one at random, or effectively clone the computer and go to all of the output states in parallel.

I've never worked with a NFSM, but they feature in the question P = NP?, as I describe in my article about it.

Hidden Markov Models

HMMs are similar to non-deterministic FSMs. You use a HMM to model a process where all you know about the process are observations of its outputs. Based on the observations so far and the current observation, you try to deduce what stage or state that process has got to.

A HMM is similar to a FSM in that there are states and transitions. However, the relationship between transitions and observations / inputs is probabilistic rather than definite. If the process has got to state S and you observe O, then there might be a 73% chance that you should then move to state T. (There might also be a 20% chance to move to state U and a 7% chance to stay in state S.)

This might seem odd and not very useful, but I have seen them used in speech recognition with decent results. The process being modelled is the production of speech. The observations are samples of some audio signal. It could be that if you've heard "p" and then hear something that sounds like a fragment of "o" (or ɒ in the International Phonetic Alphabet), that you're just starting to hear an "o" sound, as in "pod". But the fragment of "o" might actually be the beginning of the diphthong "oy" (or ɔɪ) as in "poison".

The HMM starts at a single state, and then fans out to many different words' sounds in parallel. When the first bit of input sound arrives, it produces a different probability for each transition from the output. So, the beginning of a "p" sound would give a much higher probability for the transitions that lead to words starting with p, a lower probability for transitions starting with b (because p and b are related in speech), and then a lower probability still for all other words' transitions. Given the large number of possible words, it's likely that you will immediately prune the low-probability words from consideration.

As more and more speech comes in, the probability for a given word will change, based on the probability of all the transitions traversed so far. The pruning process continues, so that the number of possibilities can be kept as manageable as possible. The pruning process can also be helped by adding in extra information. For instance, if the last word recognised was "the", then it's very unlikely (based on common English usage) that the next word would be "with", "of" etc. So you can prune those more aggressively. Similarly, if "orange" has appeared in the last few minutes, you can use that to favour "orange" over e.g. "octopus" if you have just recognised an "o" sound.

I think that HMMs are no longer at the cutting edge of speech recognition technology. Instead it's more likely that you would use a neural network, for instance a recurrent neural network.

Related

Comparing regular expressions and finite state machines
January 1, 2022
In "Computer theory"

Fuzzy matching – example algorithms
April 12, 2023
In "Computer theory"

Fault tolerance
March 14, 2018
In "Links"

‹ PREVIOUS
Regular expressions

NEXT ›
Comparing regular expressions and finite state machines

Recent Posts