

Finite State and Turing Machines



From slot machines to traffic lights, many tasks can be modelled by Finite State Machines. And when they won't do, learn how to build a Turing Machine

A coin machine is designed to dispense a ticket when sufficient coins of any denomination (and in any order) are inserted. A subroutine 'greps' through a file, stopping each time it finds a word containing a particular sequence of characters (see *Grepping your way through a record*). A program stops running when a secret sequence of digits – acting like a 'combination lock' – is keyed within a certain time interval.

These functions can all be performed by an elegant little programming device called a Finite State Machine (FSM). An FSM is actually little more than a skeleton that becomes fleshed out when data is fed into it. It can be a physical object, but here in the *Workshop* we are more concerned with its actual working, so we shall be looking at pieces of computer code.

A Finite State Machine is treated like a 'black box': it takes one signal as input and gives one signal for output. Once built, you depend on it to work in its prescribed manner, but you don't need to stop and think how.

The distinctive thing about an FSM is that it can change the way it works when you feed it an input signal. For example, suppose you have an FSM that has three lamps on it, one red, one amber and one green.

Let's say the green lamp is on and the other two are off. When you push a button (the input signal) this FSM functions, as planned, by turning off the green lamp and turning on the amber lamp. But pushing the button also changed it internally: it now works differently

because the next push of the button turns off the amber lamp and turns on the red lamp. But guess what? With the second push of the button, the internal state changed again: now it is primed so that the next push will turn on the amber lamp. Next time it will extinguish both the red and the amber, turn on the green and return to its original internal state.

More than just pretty lights

Traffic lights are an example of a fairly simple FSM; there is only one stimulus, the signal to change to the next phase. There are other complications, such as how long each phase is, and there may, of course, be additional phases – advanced green, for example. But such complications are minor and easily encompassed within the FSM structure by modifying the way input stimulus is measured and perhaps adding new states.

When you are designing an FSM, an easy way to handle the relationships between states is to build a network diagram. Each node represents a state. Nodes are connected with 'edges' (to use the normal network terminology), indicating a change between states. Every edge should have an arrow indicating direction. On the start of each connecting edge, just coming out of the node, we can record what the input stimulus was that would trigger this particular change in state. In the case of traffic lights, the output associated with each state is the configuration of illuminated lamps.



SEE OUR **SUPERDISC**

PATH: \handson\wilf.htm.

The hesitant computer

It is much more common that each state of an FSM will be able to recognise more than one input. For the sake of simplicity, let's build a Finite State Machine that will take, in any state, either a '0' or a '1' as input. From its State Transition Network can you work out how this 'hesitation' FSM in the illustration to the right works?

In this FSM, and generally, the output is seen each time there is a new input. When you look at the 'black box' you cannot see the workings so you will not necessarily know which State prevails. When you input a character, whether a '0' or '1', you will learn that the output from the State you are leaving is '0'. If you had keyed a '0', you will return to that same State, but if you had keyed a '1', the machine would have flipped over into State 'b'.

Similar choice occurs from State 'b', where keying a '0' will output '1' and then move into State 'a', but keying a '1' will output '1' and stay in State 'b'. The net effect of this is that this pair of States, linked up as they are, form a 'black box' that remembers each input and spews it out when the next input arrives – a slight delay, or hesitation. It shouldn't take much ingenuity to think of where a delay mechanism could be very useful. But is there a way to delay for the length of time it takes to do two input signals, or three?

THE GREP TEST

Grepping your way through a record

One of the most interesting applications for an FSM is the GREP function

UNIX operators are familiar with a useful command, G (for 'global') followed by a regular expression, followed by P (for 'print') – Grep for short, where 're' is where the regular expression goes. A regular expression can look like this:

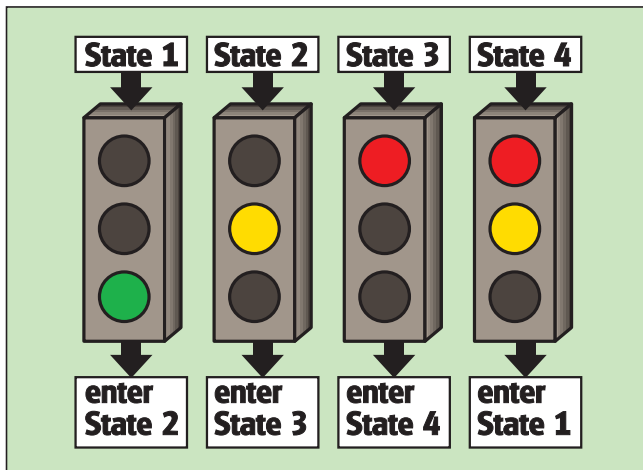
```
d.a[a-z^q]
```

Fed into the GREP function, this will begin a high-speed text search through a record or file for a four-byte string made up of a digit (indicated by

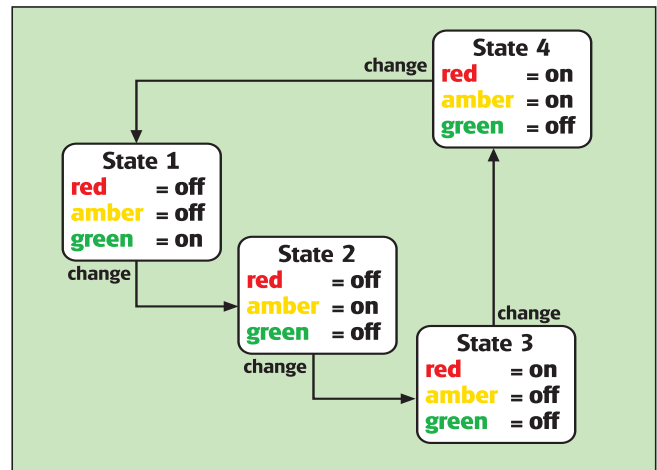
the 'd'), any character at all (indicated by the '^'), a letter of the alphabet (indicated by the first 'a'), and finally any letter of the alphabetic besides 'q' (indicated by '[a-z^q]'). You can see that this powerful technique enables you to search using very specific criteria.

It is all accomplished quite simply. The GREP function sets up several states with their own test and links them together to form a Finite State Machine. In this instance, the first state will ignore each input byte, one by one, until it finds a digit (indicated by 'd'). When it detects a digit, the

specially-built FSM makes a transition to the next internal state, waiting for an input character of any value (indicated by '^') as the second character. When this happens, the FSM enters the next state and waits for a letter of the alphabet in the next byte (indicated by 'a'). The 'machine' will only terminate the search and go on to the PRINT stage when all four of the states have been satisfied. Think about the FSM in a GREP routine as a little combination lock, which will not acknowledge finding a string match until each of the conditions is met, and in the proper order.



A set of traffic lights is actually a Finite State Machine with four internal states: the input stimulus (signal to change) produces different results (pattern of lights) for each of the states, and determines the next state entered.



The State Transition Network (STN) for the traffic lights FSM. This is a very simple machine because there is only one input to change each state, and the sequence of states is a repeated unchanging pattern.

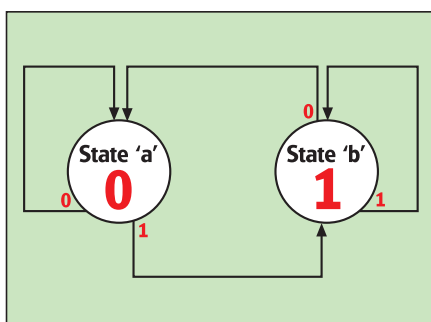
The answer to both is a resounding yes – for any length delay. Let's consider a delay of three input signals. If our input sequence is 01100011101, we want an output XXX01100011, where we ignore the first three outputs. We expect that the next three will be 101, whatever the input. How many nodes (States) will be needed? How complex will this FSM have to be?

Here's the trick to designing any FSM that has to have a 'memory' about its input: create as many States as there are possible patterns to memorise. In the case of a three-signal delay, there are eight possibilities between 000 and 111 (treat them as binary numbers and it will become obvious).

Assign each possibility to a state and then work on the basis that the rightmost bit is the output from that particular State. Of course this means that four different States are each going to output a '0', while the other four output a '1'. Whenever an input is made, that digit is added on the left side of the string associated with the current State at the same time you drop the rightmost digit (as it becomes output). If State 'a' is assigned the string 000, an input of '1' will swing it into State 'e', representing string 100. You will find that delay like these can be reorganised into pleasingly symmetric networks.

This method will apply to the construction of an FSM with any desired length of delay.

A slightly more complicated FSM, the 'hesitation' routine feeds back the previous input each time a new input comes. It needs two edges from each mode, one for each possible output character.



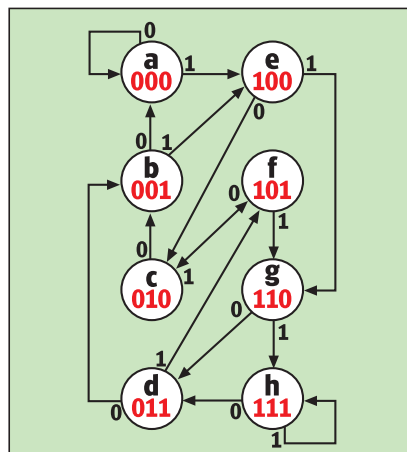
As an exercise, try mapping out the States needed for a coin machine that works in the following way: it will take British coins of the following values: 5p, 10p, 20p and 50p. It will accept each coin, and when the total is exactly 60p it will dispense a ticket. In any State it can tell the value of the coin inserted and it can reject the coin if it does not recognise it (or if it would exceed the 60p total). Of course, the machine should be able to accept the coins in any order.

Finite limits

Versatile as the Finite State Machine is, there are many things it cannot do. But a little changing around of concepts and a pair of new rules, and we end up with a machine as powerful as any supercomputer. There's a catch: we are not talking about how many millions of instructions are processed per second – we are talking about capability of doing a calculation or other task.

Any Finite State Machine is a drudge. Its workload is predictable, and the whole activity of transition between States is predetermined.

This Finite State Machine internally 'remembers' a sequence of three binary digits. As you key in each new binary digit, it spews forth the one keyed three strokes before. Larger FSMs with more States will remember even longer strings.



Furthermore, there are things it just cannot do. For example, there isn't a FSM in existence that can count indefinitely. Sure, you can build a FSM that works fine as an odometer. It will be able to count up to a limit – maybe 999,999 – but it just cannot break that one fine point and learn that advancing to a million works on the same principle as advancing to a thousand.

Want to balance parentheses in an equation string? An FSM will never do it. We can construct an FSM to do marvellous things – like add two numbers (up to a predefined length) – but try as we might, we will never be able to build an FSM that can multiply those same two numbers.

On to the Turing Machine

The changes are slight but the new supercomputer that is built on the ashes of the Finite State Machine is powerful indeed – and lots of fun to program. On the *SuperDisc* this month, we will unveil the improved model – the Turing Machine, named after its designer, the famous British computer theorist, Alan Turing. **PCP**

NEXT MONTH

How Linkages and Vectors can speed your searches and sorts.

CONTACT

How to contact Wilf

I'm always pleased to receive letters and e-mail with programming queries, ideas and opinions. As a strict rule I can't reply directly with personal one-to-one programming advice, but your input could form the basis of a future Workshop.

You can e-mail me at whew@futurenet.co.uk or write to:

Wilf's Programmers' Workshop,
PC Plus, Future Publishing,
30 Monmouth Street,
Bath BA1 2BW.
● Wilf Hey

