International Journal of
Emerging Research in Management &Technology
ISSN: 2278-9359 (Volume-6, Issue-6)

Research Article | **June 2017**

A Discussion of Finite State Machine in String Search Based on Function & Class FSM Implementations

Lawerence M.O
Computer Science Department,
Baze University, Abuja, Nigeria

Adewole O.O
Gamma Hi-Fi Electro Acoustics Company,
Port Harcourt, Nigeria, &Abidjan, Costa d'Avorio

Abstract:

The simplest type of computing machine that is worth considering is called a 'finite state machine'. As it happens, the finite state machine is also a useful approach to many problems in software architecture, only in this case you don't build one you simulate it. Essentially a finite state machine consists of a number of states – finite naturally! When a symbol, a character from some alphabet say, is input to the machine it changes state in such a way that the next state depends only on the current state and the input symbol. Notice that this is more sophisticated than you might think because inputting the same symbol doesn't always produce the same behaviour or result because of the change of state. A few examples based on the C++ implementation of the finite state algorithm based on the function & class objects is presented.

Key words: finite state machine: FSM, String, function, class FSM, string matching.

I. INTRODUCTION

Finite state machines

A finite state machine (FSM, also known as a deterministic finite automaton or DFA) is a way of representing a *language* (meaning a set of strings; we're interested in representing the set strings matching some pattern).

The problem of determining the language accepted by a given finite state acceptor is an instance of the algebraic path problem* —itself a generalization of the shortest path problem to graphs with edges weighted by the elements of an (arbitrary) semiring (Storer, 2001,Pouly, 2011).

It's explicitly algorithmic: we represent the language as the set of those strings *accepted* by some program. So, once you've found the right machine, you can test whether a given string matches just by running it.

The KMP algorithm works by turning the pattern it's given into a machine, and then running the machine. The hard part of KMP is finding the machine.

We need some restrictions on what we mean by "program". This is where "deterministic & finite" come from.

One way of thinking about it, is in terms of programs without any variables. All such a program can do is look at each incoming character determine what line to go to, and eventually return true or false (depending on whether it thinks the string matches or doesn't).

The simplest type of computing machine that is worth considering is called a 'finite state machine'.

As it happens, the finite state machine is also a useful approach to many problems in software architecture, only in this case you don't build one you simulate it.

Essentially a finite state machine consists of a number of states – finite naturally! When a symbol, a character from some alphabet say, is input to the machine it changes state in such a way that the next state depends only on the current state and the input symbol.

Notice that this is more sophisticated than you might think because inputting the same symbol doesn't always produce the same behaviour or result because of the change of state.The new state depends on the old state and the input.

What this means that the entire history of the machine is summarized in its current state. All that matters is the state that it is in and not how it reached this state. Before you write off the finite state machine as so feeble as to be not worth considering as a model of computation it is worth pointing out that as you can have as many states as you care to invent the machine can record arbitrarily long histories. All you need is a state for each of the possible past histories and then the state that you find the machine in is an indication of not only its current state but how it arrived in that state.

Because a finite state machine can represent any history and a reaction, by regarding the change of state as a response to the history, it has been argued that it is a sufficient model of human behaviour i.e. humans are finite state machines.

If you know some probability theory you will recognize a connection between finite state machines and Markov chains. A Markov chain sums up the past history in terms of the current state and the probability of transition to the next state only depends on the current state. The Markov chain is a sort of probabilistic version of the finite state machine.

As a simple warmup example, let's look at a program for an easier problem: testing whether a string has an even number of characters.

```
main()
{
   for (;;) {
```



```
        if (getchar() == EOF) return TRUE;
        if (getchar() == EOF) return FALSE;
    }
}
```

Note the lack of variables. To simplify things, we'll rewrite programs to avoid complicated loops, and instead just use goto statements. (You've probably been taught that gotos are bad, but this sort of rewriting happens all the time, in fact every time you run a compiler it has to do this.)

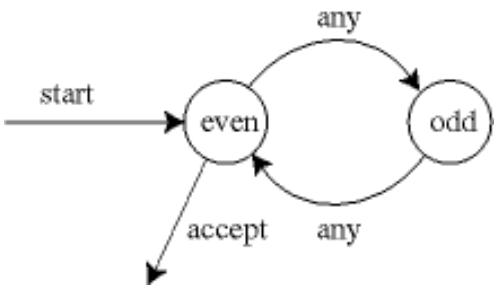
```
main()
{
    even:
        if (getchar() == EOF) return TRUE;
        else goto odd;

    odd:
        if (getchar() == EOF) return FALSE;
        else goto even;
}
```

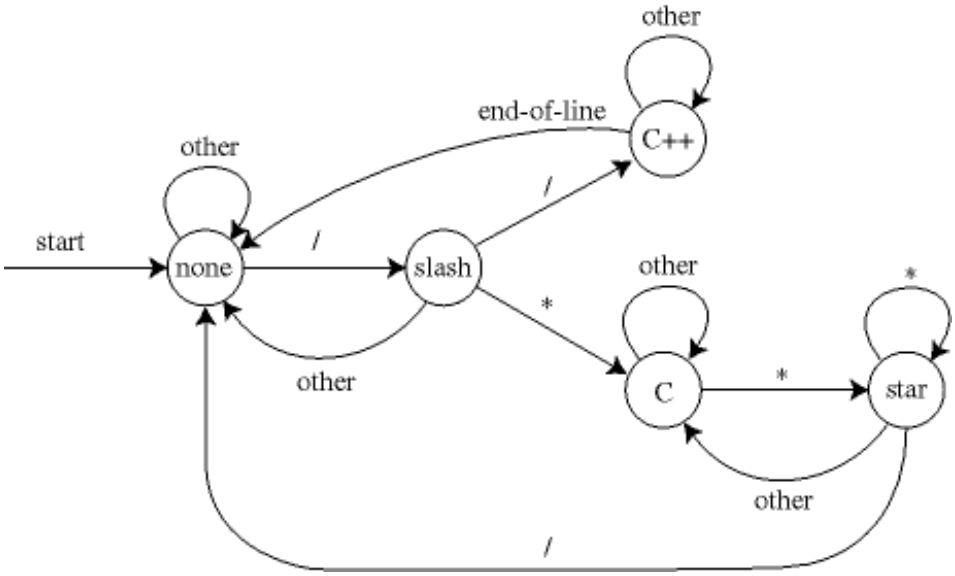
We've chosen labels for the goto statements, to represent what we know about the string so far (in this problem, whether we've seen an even or odd number of characters so far). Because there are no variables, we can only represent knowledge about the input in terms of where we are in the program. We think of each line in the program as being a *state*, representing some specific fact about the part of the string we've seen so far. Here the states are "even" and "odd", and represent what we know about the number of characters seen so far.

Since there are no variables, the only thing a machine can do in a given state (state = what line the prog is on) is to go to different states, depending on what character it sees.

This can be a useful programming style; for instance in using a program writen in close to this style** to filter some html files on a web pages. One advantage of this style is that there are few ways the program can be tricked into having unexpected values in its variables (since there are no variables) so it is hard to make such a program crash. But it's a little long and cumbersome, and you wouldn't want to have to compile a separate C program every time you ran "grep". Rather than writing C code, we'll draw pictures with circles and arrows. (These pictures are known as *state diagrams*.) A circle will represent a state, an arrow with a label will represent that we go to that state if we see that character. (You can think of this as just being a special kind of graph.) We'll also draw an arrow from nowhere to the first state the program starts in, and arrows to nowhere if the program returns true if the string ends at that state. So our program can be represented with the following diagram.



In class I described a more complicated example, that could be used by the *C preprocessor* (a part of most C compilers) to tell which characters are part of comments and can be removed from the input:



It's easy to turn such a diagram into a program, that simply has one label and one case statement per state.

If we're given such a diagram, and a string, we can easily see whether the corresponding program returns true or false. Simply place a marker (such as a penny) on the initial state, and move it around one state at a time until you run out of characters. Once you run out of characters, see whether the state you're in has an "accept" arrow -- if so, the pattern matches, and if not it doesn't.

In a computer, we obviously don't represent these things with circles and arrows. Instead they can be viewed as just being a special kind of graph, and we can use any of the normal graph representation*to store them.

One particularly useful representation is a *transition table*: we make a table with rows indexed by states, and columns indexed by possible input characters. Then simulating the machine can be done simply by looking up each new step in the table. (You also need to store separately the start and accept states.) For the machine above that tests whether a string has even length, the table might look like this:

	any
even:	odd
odd:	even

For the C comment machine, we get a more complicated table:

	/	*	EOL	other
none:	slash	none	none	none
slash:	C++	C	none	none
C++:	C++	C++	none	C++
C:	C	star	C	C
star:	none	star	C	C

Since a state diagram is just a kind of graph, we can use graph algorithms to find some information about finite state machines. For instance we can simplify them by eliminating unreachable states, or find the shortest path through the diagram (which corresponds to the shortest string accepted by that machine).

II. AUTOMATA AND STRING MATCHING

The examples above didn't have much to do with string matching. Let's look at one that does. Suppose we want to "grep nano". Rather than just starting to write states down, let's think about what we want them to mean. At each step, we want to store in the current state the information we need about the string seen so far. Say the string seen so far is "...stuvwxy", then we need to know two things:

1. Have we already matched the string we're looking for ("nano")?
2. If not, could we possibly be in the middle of a match?

If we're in the middle of a match, we need to know how much of "nano" we've already seen. Also, depending on the characters we haven't seen yet, there may be more than one match that we could be in the middle of -- for instance if we've just seen "...nan", then we have different matches if the next characters are "o..." or if they're "ano...". But let's be optimistic, and only remember the longest partial match.

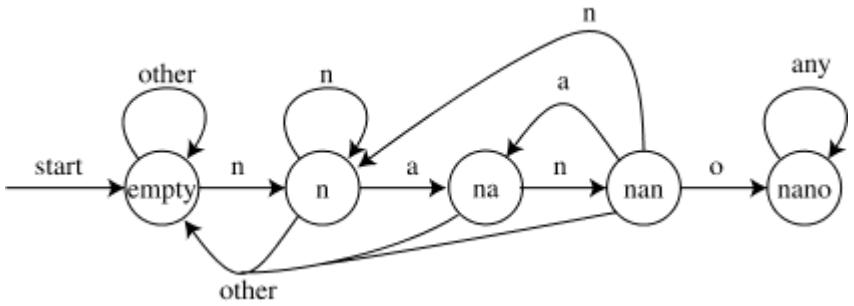
So we want our states to be partial matches to the pattern. The possible partial matches to "nano" are "", "n", "na", "nan", or (the complete match) "nano" itself. In other words, they're just the *prefixes* of the string. In general, if the pattern has m characters, we need m+1 states; here m=4 and there are five states.

The start and accept states are obvious: they are just the 0- and m-character prefixes. So the only thing we need to decide is what the transition table should look like. If we've just seen "...nan", and see another character "x", what state should we go to? Clearly, if x is the next character in the match (here "o"), we should go to the next longer prefix (here "nano"). And clearly, once we've seen a complete match, we just stay in that state. But suppose we see a different character, such as "a"? That means that the string so far looks like "...nana". The longest partial match we could be in is just "na". So from state "nan", we should draw an arrow labeled "a" to state "na". Note that "na" is a prefix of "nano" (so it's a state) and a *suffix* of "nana" (so it's a partial match consistent with what we've just seen).

In general the transition from state+character to state is the longest string that's simultanously a prefix of the original pattern and a suffix of the state+character we've just seen. This is enough to tell us what all the transitions should be. If we're looking for pattern "nano", the transition table would be

	n	a	o	other
empty:	"n"	empty	empty	empty
"n":	"n"	"na"	empty	empty
"na":	"nan"	empty	empty	empty
"nan":	"n"	"na"	"nano"	empty
"nano":	"nano"	"nano"	"nano"	"nano"

For instance the entry in row "nan" and column n says that the largest string that's simultaneously a prefix of "nano" and a suffix of "nan"+n="nann" is simply "n". We can also represent this as a state diagram:



Simulating this on the string "bananonona", we get the sequence of states empty, empty, empty, "n", "na", "nan", "na", "nan", "nano", "nano", "nano". Since we end in state "nano", this string contains "nano" in it somewhere. By paying more careful attention to when we first entered state "nano", we can tell exactly where it occurs; it is also possible to modify the machine slightly and find all occurrences of the substring rather than just the first occurrence.

This description is enough to get a string matching algorithm that takes something like O(m^3 + n) time: O(m^3) to build the state table described above, and O(n) to simulate it on the input file. There are two tricky points to the KMP algorithm. First, it uses an alternate representation of the state table which takes only O(m) space (the one above could take O(m^2)). And second, it uses a complicated loop to build the whole thing in O(m) time. We'll see this algorithm next time.

III. CONCEPT

Deterministic finite automation algorithm

*:

For a DFA M = (Q, Σ, δ, q0, F), we say that a state q ∈ Q is reachable if there exists some string w ∈ Σ∗ such that q = δ∗(q0, w).

Give an algorithm that, given as input a DFA expressed as a five-tuple M = (Q, Σ, δ, q0, F), returns the set of all of M's reachable states.

Here, I provide C++ implementation of a table-based Deterministic Finite Automaton. A DFA is a finite state machine that accepts/rejects finite strings of symbols and has a unique transition from each state on each input symbol.
A deterministic finite automaton M is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, consisting of

- a finite set of states (Q)
- a finite set of input symbols called the alphabet (Σ)
- a transition function ($\delta : Q \times \Sigma \rightarrow Q$)
- a start state ($q_0 \in Q$)
- a set of final/accept states ($F \subseteq Q$)

Let $w = a_1 a_2 \ldots a_n$ be a string over the alphabet Σ. The automaton M accepts the string w if a sequence of states, $r_0, r_1, \ldots, r_n$, exists in Q with the following conditions:

1. $r_0 = q_0$
2. $r_{i+1} = \delta(r_i, a_{i+1})$, for $i = 0, \ldots, n-1$
3. $r_n \in F$.

The first condition says that the machine starts in the start state q_0. The second condition says that given each character of string w, the machine will transition from state to state according to the transition function δ. The last condition says that the machine accepts w if the last input of w causes the machine to halt in one of the accepting states. Otherwise, it is said that the automaton *rejects* the string. The set of strings M accepts is the language *recognized* by M and this language is denoted by $L(M)$.

Take a look at the C++ implementation. It requires a single pass over the input strings to decide whether or not they belong to the language of a given DFA.

IV. DISCUSSION

In string matching:

You probably often use your text editor (or the UNIX program "grep") to find some text in a file (e.g. the place where you defined your depth first search program, or the email message you sent six weeks ago asking for an extension on your programming project).
How does it do it?

There are two commonly used algorithms: Knuth-Morris-Pratt (KMP) and Boyer – Moore (BM). Both use similar ideas. Both take linear time: O(m + n) where m is the length of the search string, and n is the length of the file. Both only test whether certain characters are equal or unequal, they don't do any complicated arithmetic on characters.

Boyer-Moore is a little faster in practice, but more complicated. Knuth-Morris-Pratt is simpler, could be preferrably discussed in terms of finite state machines.*'

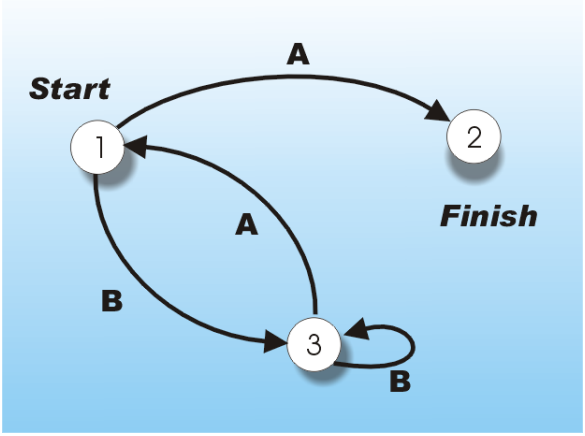
A finite state machine with only one state is called a "combinatorial FSM". It only allows actions upon transition *into* a state. This concept is useful in cases where a number of finite state machines are required to work

together, and when it is convenient to consider a purely combinatorial part as a form of FSM to suit the design tools(Brutscheck, 2008). *

There are other sets of semantics available to represent state machines. For example, there are tools for modeling and designing logic for embedded controllers(Tiwari, 2012).They combine hierarchical state machines (which usually have more than one current state), flow graphs, and truth tables into one language, resulting in a different formalism and set of semantics(Hamon, 2005). These charts, like Harel's original state machines(Harel, 1987), support hierarchically nested states, orthogonal regions, state actions, and transition actions(Alur, 2008).

Examples : Implementation *'

As a simple example consider the finite state machine given earlier with state 1 as start and state 3 as finish – what sequences does it accept? Assuming that A and B are the only two symbols available, it is clear from the diagram that any sequence like BABAA is accepted by it.



A finite machine accepts a set of sequences

In general the machine will accept all sequences that can be described by the computational grammar*

1) <null> -> B<S1>|A#
2) <S1> -> A<S2>
3) <S2> -> B<S1>|A#

The only new features are the use of <null> to specify the starting state and the use of # to specify the final state. You can have many hours of happy fun trying to prove that this grammar parses the same sequences as the finite state machine accepts.

To see that it is it does just try generating a sequence:

- start with <null> and apply rule 1 to get B<S1>
- use rule 2 to get BA<S2>
- use rule 3 to get BAB<S1>

You can carry on using rule 2 and 3 alternately until you get bored and decide to use the A# alternative of rule 3 giving something like BABABABAA#.

Example 1:

Output:

I'm doing an assignment for automata theory, which I have to determine whether a word is accepted or not by a transition function for a deterministic finite automaton

I have this input file:

```
6802
2
5
00 a
01 a
11 b
12 c
13 c
34 d
44 d
45 d
3
aaabcccc
aabbbbcdc
acdddddd
```


The input starts with 4 integers, the first is the number of state for the automaton, next is the number of transitions of the automaton, the third number is the initial state, and then the number of final states. then come the final states (in the example the final states are 2 and 5).

Then come N lines (N is the number of transitions), each with 2 integers and a character, I, J and C, representing the states where the transition, ie, the transition goes from state i to state J with the character C. Following this line come with a single integer S, which will contain the number of strings to test, then S lines with the respective strings.

The output of this program should be:

```
Case#2:
aaabcccc Rejected
aabbbbcdc Rejected
acdddddd Accepted
```

It should say if the String is accepted or rejected. So far, I've only coded the work with the input.

Example 2:

Output:

```
$ g++ StringMatchingFiniteAutomata.cpp
$ a.out

 pattern found at index 0
 pattern found at index 9
 pattern found at index 13
------------------
(program exited with code: 0)
Press return to continue
```

V. CONCLUSION

The simplest type of computing machine that is worth considering is called a ‘finite state machine’.

As it happens, the finite state machine is also a useful approach to many problems in software architecture, only in this case you don’t build one you simulate it.

Essentially a finite state machine consists of a number of states – finite naturally! When a symbol, a character from some alphabet say, is input to the machine it changes state in such a way that the next state depends only on the current state and the input symbol.

Notice that this is more sophisticated than you might think because inputting the same symbol doesn’t always produce the same behaviour or result because of the change of state.

You probably often use your text editor (or the UNIX program "grep") to find some text in a file (e.g. the place where you defined your depth first search program, or the email message you sent six weeks ago asking for an extension on your programming project).

How does it do it?

There are two commonly used algorithms: Knuth-Morris-Pratt (KMP) and Boyer -Moore (BM). Both use similar ideas. Both take linear time: O(m + n) where m is the length of the search string, and n is the length of the file. Both only test whether certain characters are equal or unequal, they don't do any complicated arithmetic on characters.

Boyer-Moore is a little faster in practice, but more complicated. Knuth-Morris-Pratt is simpler.

If you define two of the machine’s states as special – a starting and a finishing state – then you can ask what sequence of symbols will move it from the starting to the finishing state.

Any sequence that does this is said to be ‘accepted’ by the machine. Equally you can think of the finite state machine as generating the sequence by outputting the symbols as it moves from state to state.That is a list of state changes obeyed in order, from the start to the finish state, generates a particular string of symbols. Any string that can be generated in this way will also be accepted by the machine.

The point is that the simplicity or complexity of a sequence of symbols is somehow connected to the simplicity or complexity of the finite state machine that accepts it.

So we now have a way to study sequences of symbols and ask meaningful questions.

REFERENCES

[1] Alur, R., Kanade, A., Ramesh, S., & Shashidhar, K. C. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. International Conference on Embedded Software,Atlanta, GA: ACM, pp. 89-98,2008.

[2] *Belzer, Jack; Holzman, Albert George; Kent, Allen. Encyclopedia of Computer Science and Technology.* ***25****. USA: CRC Press. p. 73, 1975.*

[3] Brutscheck, M., Berger, S., Franke, M., Schwarzbacher, A., Becker, S.: Structural Division Procedure for Efficient IC Analysis. IET Irish Signals and Systems Conference, (ISSC 2008), pp.18-23. Galway, Ireland, 18–19 June 2008. [1]

[4] *Hamon, G. A Denotational Semantics for Stateflow. International Conference on Embedded Software. Jersey City, NJ: ACM. pp. 164–172, 2005.*

[5] Harel, D. A Visual Formalism for Complex Systems. Science of Computer Programming , 231–274, 1987.
[6] http://www.iam.unibe.ch/~run/talks/2008-06-05-Bern-Jonczy.pdf, p. 34
[7] *John E. Hopcroft and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation. Reading/MA: Addison-Wesley, 1979.*
[8] *Koshy, Thomas. Discrete Mathematics With Applications. Academic Press. p. 762, 2004.*
[9] *Keller, Robert M. "Classifiers, Acceptors, Transducers, and Sequencers" (PDF). Computer Science: Abstraction to Implementation (PDF). Harvey Mudd College. p. 480, 2001.*
[10] *Pouly, Marc; Kohlas, Jürg. Generic Inference: A Unifying Theory for Automated Reasoning. John Wiley & Sons. Chapter 6. Valuation Algebras for Path Problems, p. 223 in particular,2011.*
[11] *Storer, J. A. An Introduction to Data Structures and Algorithms. Springer Science & Business Media. p. 337, 2001.*
[12] Tiwari, A. Formal Semantics and Analysis Methods for Simulink Stateflow Models, 2002.
[13] *Wright, David R. (2005). "Finite State Machines" (PDF). CSC215 Class Notes. David R. Wright website, N. Carolina State Univ. Retrieved July 14, 2012.*

