International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 04 Issue: 10 | Oct -2017 www.irjet.net p-ISSN: 2395-0072

Pattern Matching using Computational and Automata Theory

Vennila Santhanam

Assistant Professor,Computer Science Department, Auxilium College (Autonomous), Vellore

---

Abstract - *Automata Theory is found useful in many high-level programming languages. It can be applied for the evaluation of regular expressions. Pattern matching requires a complicated model, with a different programmatic approach. There are many techniques available for pattern matching process that is memory efficient which reduces the size of Deterministic finite automata. Finite Automata is used in pattern matching process to represent the patterns. To make it memory efficient we can minimize the number of states, minimize number of transitions. In this paper we present a new automata-based approach for pattern matching. We use a macro that takes a grammar and generates a function that reads off the leaves of a tree and tries to parse them as a string in a context-free language. The experimental results indicate that this approach is a tool for pattern matching.*

Keywords: Automata Theory; Pattern Matching; Regular Languages; Finite automata, Regular expression.

1. INTRODUCTION

Finite automata

A model of computation composed• of states, a transition function, and an input alphabet.

Finite State Machine

An automaton (in automata theory) is a 5-tuple (Q, Σ, δ, q0, F) defined as following:

- Q – Finite set of states
- Σ – Alphabet

δ – Transition function (δ: Q × Σ → Q)

- q0 – First (starting) state
- F – Set of finishing (accept) states

Pattern Matching

It is the act of checking a given sequence of tokens for the presence of the constituents of some pattern.

Transition function

It describes a condition that has to be fulfilled to enable the transition.

Input alphabet

The input recognized by the Finite State Machine

Regular Language

It is a formal language that can be expressed using a regular expression.

2. ANALYSIS OF ALGORITHMS FOR PATTERN MATCHING

An automaton is a machine that scans a string and either accepts it or rejects it. The string is accepted if the automaton reaches the finishing (accept) state after "reading" it. "Reading" the string is done one symbol at a time and using the transition function determine what the next state will be. If the automaton is not in an accept state at the end, the string gets declined. Finite automata can be divided into two subgroups. Automata can be either deterministic (DFA) or nondeterministic (NFA). DFA is deterministic; meaning the transition from one state to another is unique. In NFA transition in the automaton can go from one state to several different states by "reading" only one symbol. finite automata is usually represented by a directed graph where arrows represent the transition function.

There are two commonly used algorithms for pattern matching:

- Knuth-Morris-Pratt (KMP)
- Boyer-Moore (BM)

Both the algorithms make use of similar method. The complexity of the algorithms take linear time: O(m + n) where m is the length of the string, and n is the length of the file. The main drawback of these algorithms is that they just check whether certain characters are equal or unequal. No arithmetic operation is performed.

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

Finite state machines

A finite state machine (FSM) is used for representing a language . A language L is a set of strings. If the strings

are accepted by the FSM then the language is for the automation. We can write L(M), Where M is the FSM.

Algorithm

We represent the language as the set of those strings *accepted* by some program. Once we find the right machine, we can test whether a given string matches just by running the program.

In KMP algorithm first the pattern is turned into a machine, then run the machine. The most important and difficult 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).

A program for testing whether a string has an even number of characters.

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

There are no variables in the above program. We can avoid complicated loops, and use goto statements.

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

As 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".

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

The program can be represented using State transition 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. It is a special kind of graph. The start state is also indicated with arrow from nowhere. The program returns true if the string ends at that state. So our program can be represented with the following diagram.

Fig -1: **State transition diagram for testing whether a string has an even number of characters**

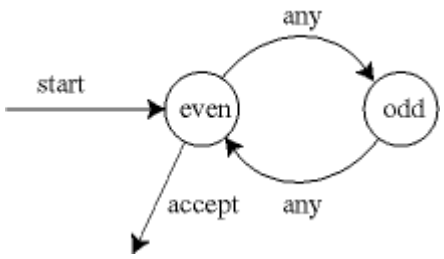
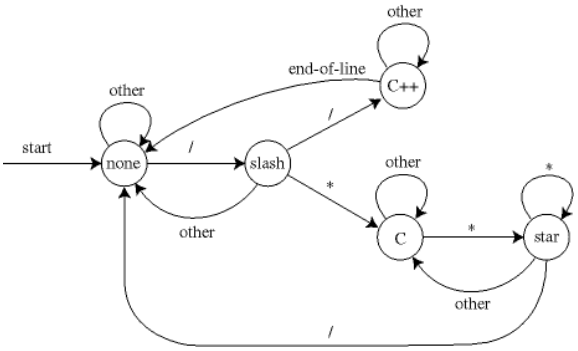


Fig -2: **State transition diagram for identifying comments in a c program**.



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 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 can use any of the normal graph representations 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:

Table -1: Transition table for determining whether a string has even length or odd length.

States/input	any
odd	even
even	odd

Table -2: Transition table for the C comment machine.

States/input	/	*	EOL	other
empty	slash	empty	empty	empty
C++	C++	C++	empty	C++
asterisk	empty	asterisk	C	C
slash	C++	C	empty	empty
C	C	asterisk	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.

Automata and string matching

If we want to match "automata". 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 ("auto")?
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 "mata" we've already seen.

Depending on the characters we haven't seen yet, there may be more than one match that we could be in the middle.

So we want our states to be partial matches to the pattern. The possible partial matches to "memo" are "", "m", "me", "mem", or (the complete match) "memo" 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.

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 "memo", the transition table would be:

Table -3: Transition table for the pattern "memo".

States /input	m	e	o	other
empty	"m"	empty	empty	empty
"m"	"m"	"me"	empty	empty
"me"	"mem"	empty	empty	empty
"mem"	"m"	"me"	"memo"	empty
"memo"	"memo"	"memo"	"memo"	"memo"

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

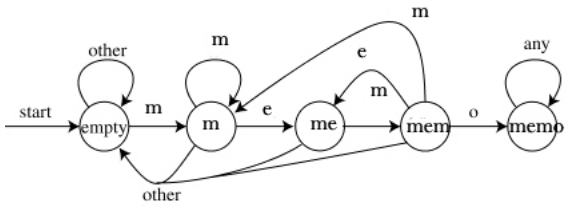


Fig -3: State transition diagram for the pattern "memo".

Simulating this on the string "bananamemo", we get the sequence of states empty, empty, empty, "m", "me", "mem", "me", "mem", "memo", "memo", "memo". Since we end in state "memo", this string contains "memo" in it somewhere. By paying more careful attention to when we first entered state "memo", 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.

3. EXPERIMENT USING THE STRING "BARBARA"

"barbara" is a fun word. It consists of just three letters (b, a, r) and has the substring bar repeated twice. Given a random string of text, how can we determine if barbara appears in it? The searching algorithms Knuth–Morris–Pratt, Boyer-Moore and many others may not be enough. We can make an effective search by constructing a DFA for it. The DFA will accept a string if and only if it is in an accept state after "reading" the string. Automaton given on

the graph below will be left in an accept state iff it contains barbara as a substring. Let's get on with the construction. In order to detect barbara, we will need 8 states and only one of them will be an accept state (first seven will be for checking what precedes what). So, our set Q will be Q = {0, 1, 2, 3, 4, 5, 6, 7}. For the alphabet Σ, we will use the Standard English alphabet. Strings over Σ are all words that can be generated using letters of the English alphabet (labeled as Σ*). The transition function δ is represented by the arrows on the graph below. Our starting state will be 0, and accept state set will be F = {7}. Desiging δ is the hardest part of the process, of course.

The transition function displayed on the graph is the following:

δ(3, b) = 4; // bar -> barb
δ(6, b) = 4; // barbar -> barb
δ({0, s}, b) = 1; // x -> b
δ(1, a) = 2; // b -> ba
δ(4, a) = 5; // barb -> barba
δ(6, a) = 7; // barbar -> barbara (the end)
δ(s, a) = 0; // x -> ε (empty word)
δ(2, r) = 3; // ba -> bar
δ(5, r) = 6; // barba -> barbar
δ(s, r) = 0; // x -> ε
δ(s, x) = 0; // x -> ε

s is a variable state not included in the definition
As you can see, the automaton will be in the finishing state iff it recognizes barbara as a substring. When it gets to the finishing state, the loop will make it stay there. Let's try it out and see how it works. For the example, I'll use the transition function and I'll show you each step. Before that, I have to state that δ(s, abc) = δ(δ(s, a), bc), meaning that the word can be broken at any place and we will still get the same results.

δ(0, oifsfscnbarbakjkjibarbarabkf) =
δ(δ(0, oifsfscn), barbakjkjibarbarabkf) =
δ(0, barbakjkjibarbarabkf) =
δ(δ(0, bar), bakjkjibarbarabkf) =
δ(3, bakjkjibarbarabkf) =
δ(δ(3, ba), kjkjibarbarabkf) =
δ(5, kjkjibarbarabkf) =
δ(δ(5, kjkji), barbarabkf) =
δ(0, barbarabkf) =
δ(δ(0, barbara), bkf) =
δ(7, bkf) = 7 which is an accept state.

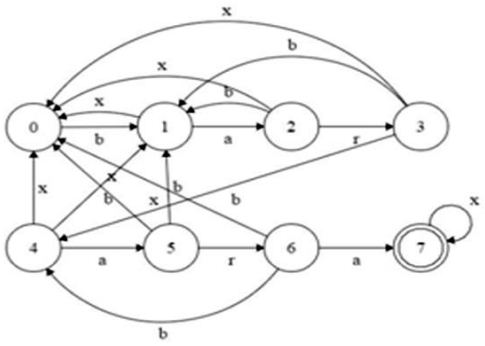


Fig -3: State transition diagram for identifying comments Deterministic Finite Automata for "barbara" Problem.

4. CONCLUSIONS

It is difficult to design pattern matching algorithms, but Finite automata can be used to match strings and regular expressions of all kinds. A Finite Automation accepts regular languages and a language is regular iff it has a regular expression representing it. The study of formal grammar and regular expressions has shown us with those topics the utility, robustness, and sometimes elegance of regular languages. The same approach can also be applied to variety of other functional programming languages. Finally, the use of automata as a symbolic representation for verification has been investigated in other contexts. Based on the pattern and its length the size of the Finite Automata may vary. The Deterministic Finite Automata possibly constructed from the Nondeterministic Finite Automata.

REFERENCES

[1] http://en.wikipedia.org/wiki/ Automata_theory

[2] Hopcraft J E, Motwani R and Ullman J D [2001], "Introduction to Automata Theory, Languages and Computation", AddisonWesley second edition.

[3] Mindek, M., "Finite State Automata and Image Recognition" DATESO 2004, pp 132-143 (2004), ISBN: 80-248-0457-3

[4] G. Navarro, R. Baeza-Yates, "Improving an Algorithm for Approximate String Matching.",Algorithmica, 30(4) 2001

[5] M. Crochemore, T. Lecroq, "Pattern Matching and Text Compression Algorithms", The Computer Science and Engineering Handbook, A.B. Tucker, Jr, ed., CRC Press, Boca Raton, 2003, Chapter 8.

[6] D. Perrin, "Finite Automata", Handbook of Theoretical Computer Science. Elsevier Science Published 1990.

[7] A.V. Aho and M.J. Corasick. —Efficient String Matching: An Aid to Bibliographic Search.‖ Communications of the ACM, 18(6):333–340, 1975.

[8] S. Kumar, B. Chandrasekaran, J. Turner, G. Varghese, —Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia‖, in Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS), pages 155-164. ACM, 2007.

[9] R. Smith, C. Estan, and S. Jha, —Xfa: Faster signature matching with extended automata‖, in IEEE Symposium on Security and Privacy, May 2008.

[10] D.Ficara, S.Giordano, G. Procissi, F.Vitucci, G.Antichi, A.D. Pietro, —An Improved DFA for Fast Regular Expression Matching‖ ACM SIGCOMM Computer Communication Review, Volume 38, Number 5, October 2008.

[11] https://www.ics.uci.edu/~eppstein/161/960222.html