An Effective Finite State Simulator for Pattern Matching Using Moore's Law

[1]Mr.Shreenath Waramballi, [2]Dr.Guruprakash C.D, [3]Mr.Sunil B N

[1]Assistant Professor, [2]Professor, [3]Assistant Professor
[1]Computer Science and Engineering,
[1]Sahyadri College of Engineering and Management, Mangalore, India

Abstract: **Pattern matching is an important phenomenon which is has most significant impact in text mining process. It is important to study the behavior of system before implementing any algorithm. A study has to be carried out carefully by imitating the behavior of the present system with respect to different inputs of different length. In this paper we are going to propose an algorithm to retrieve the given data and to store the important information using the moore's law. As the given data is to be processed character by character we prefer finite state simulator to process the data sequentially. The proposed algorithm helps to retrieve various types of information like textual sequences, biological data etc.**

IndexTerms **–Automata, Behaviour of system, moore's law, finite state simulator**

I. INTRODUCTION

Pattern matching is the process of checking a given **sequence** of symbols for the presence of the constituents of some pattern. In contrast to pattern recognition, the match usually has to be exact. The patterns generally have the form of either sequences or tree structures. Uses of pattern matching include outputting the locations of a pattern within a token sequence, to output some component of the matched pattern, and to substitute the matching pattern with some other token sequence (Sequence patterns (e.g., a text string) are often described using regular expressions and matched using techniques such as backtracking. Tree patterns are used in some programming languages as a general tool to process data based on its structure, e.g., Haskell, ML, Scala and the symbolic mathematics language Mathematics have special syntax for expressing tree patterns and a language construct for conditional execution and value retrieval based on it. For simplicity and efficiency reasons, these tree patterns lack some features that are available in regular expressions. Often it is possible to give alternative patterns that are tried one by one, which yields a powerful conditional programming construct. Pattern matching sometimes includes support for guards.

The intuitive notions of computation and *algorithm* are central to mathematics. Roughly speaking, an algorithm is an explicit, step-by-step procedure for answering some question or solving some problem. An algorithm *provides* routine mechanical instructions dictating how to proceed at each step. Computational complexity is a branch of the theory of computation that focuses on classifying computational problems according to the inherent difficulty, and relating those classes to each other. A computational problem is understood to be a task that is in principle amenable to being solved by a computer, which is equivalent to stating that the problem may be solved by mechanical application of mathematical steps, such as an algorithm. A problem is regarded as inherently difficult if its solution requires significant resources, whatever the algorithm used. The theory formalizes this intuition, by introducing mathematical models of computation to study these problems and quantifying the amount of resources needed to solve them, such as time and storage. Other complexity measures are also used, such as the amount of communication the number of gates in a circuit and the number of processors.

Simulation is the imitation of the operation of a real-world process or system over time. The act of simulating something first requires that a model be developed; this model represents the key characteristics, behaviours and functions of the selected physical or abstract system or process. The model represents the system itself, whereas the simulation represents the operation of the system over time. Simulation is used in many contexts, such as simulation of technology for performance optimization, safety engineering, testing, training, education, and video games. Often, computer experiments are used to study simulation models. Simulation is also used with scientific modelling of natural systems or human systems to gain insight into their functioning, as in economics. Simulation can be used to show the eventual real effects of alternative conditions and courses of action. Simulation is also used when the real system cannot be engaged, because it may not be accessible, or it may be dangerous or unacceptable to engage, or it is being designed but not yet built, or it may simply not exist.

Key issues in simulation include acquisition of valid source information about the relevant selection of key characteristics and behaviours, the use of simplifying approximations and assumptions within the simulation, and fidelity and validity of the simulation outcomes. Procedures and protocols for model verification and validation are an ongoing field of academic study, refinement, research and development in simulations technology or practice, particularly in the field of computer simulation.

A finite state automaton is an abstract machine that successively reads each symbols of the input string, and changes its state according to a particular control mechanism. If the machine, after reading the last symbol of the input string, is in one of a set of particular states, then the machine is said to accept the input string. It can be illustrated as follows:

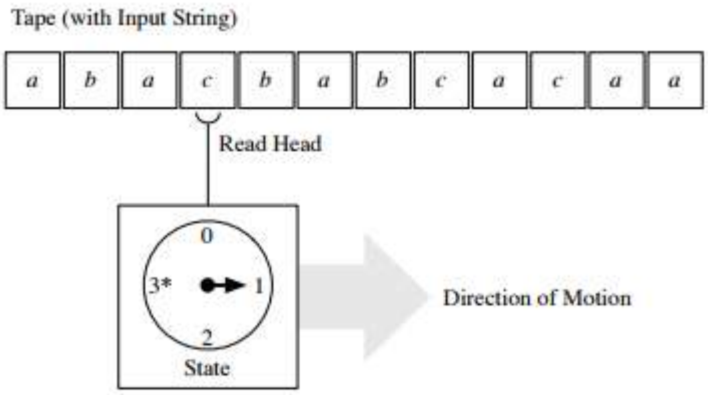
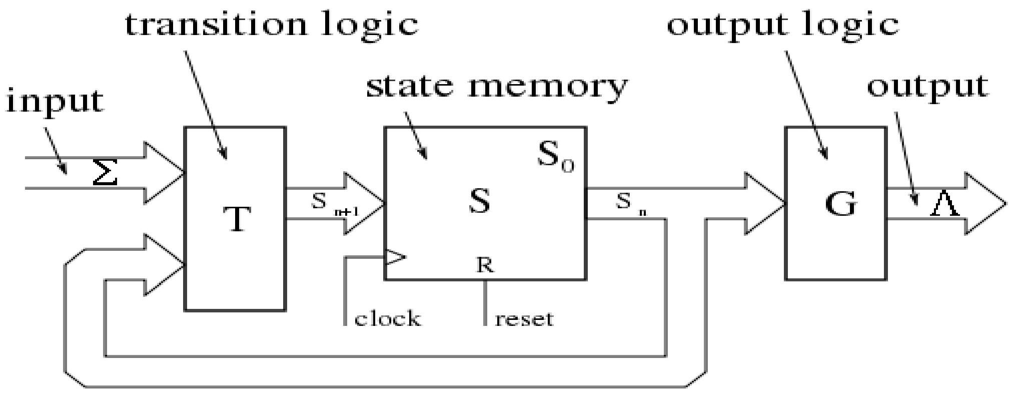


Fig:Illustration of finite state automaton

Once we created a finite state machine to solve a problem,we may wanto simulate its execution.This can be achieved using finite state simulators. A finite state transducer consists of a number of states which are linked by transitions labeled with an input/output pair. The FST starts out in a designated start state and jumps to different states depending on input while producing output according its transition table.One simple kind of state transducer associates an output with each state of a machine M. The output is generated whenever the M enters the associated state.

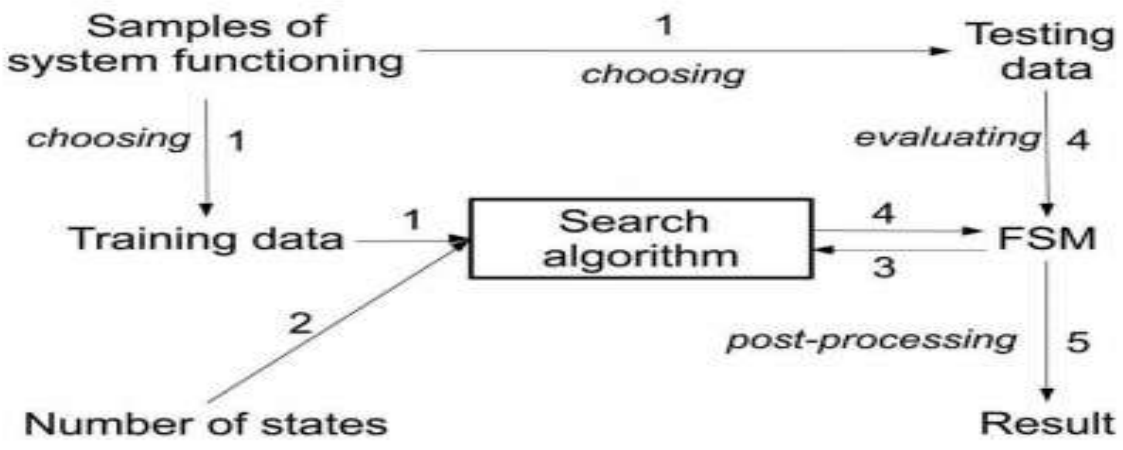
Moore's Machine-a Moore machine is a finite-state machine whose output values are determined only by its current state. This is in contrast to a Mealy machine, whose output values are determined both by its current state and by the values of its inputs**.**
A moore machine M can be defined as a 6 tuple consisting of the following M ={ S, S_0, ∑, ʌ, T, G}
1.A finite set of states S
2. a start state S_0
3. a finite set called the input alphabet ∑
4. a finite set called the output alphabet ʌ
5. A transition function T:S*∑→S
6. an output function G:S→ ʌ



A deterministic finite automaton (DFA) is a 5-tuple (Q, Σ, δ, q0, F)
where Q is a finite set called the states,
Σ is a finite set called the alphabet,
δ : Q × Σ → Q is the transition function,
q0 ∈ Q is the start state, and F ⊆ Q is the set of accept states.
A DFA can be seen as a special case of a Moore machine, where the set of input symbols I is Σ, and the set of output symbols is binary, say O = {0, 1}, with 1 and 0 corresponding to accepting and non-accepting states, respectively. The concepts of complete and incomplete DFAs, as well as the definition of δ ∗ , are similar to the corresponding ones for FSMs. Elements of Σ∗ are usually called words. A DFA A = (Σ, Q, q0, δ, F) is said to accept a word w if δ ∗ (q0, w) ∈ F.

Input-output traces and examples Given sets of input and output symbols I and O, respectively, a Moore (I, O)-trace is a pair of Syntactic Pattern Recognition (SPR) is currently seen as a very appealing approach to many Pattern Recognition (PR) problems for which the most traditional Decision-Theoretic or Statistical approaches fail to be effective . One of the most interesting features of SPR consists of the ability to deal with highly structured (representations of) objects and to uncover the underlying structure by means of parsing. In fact, it is often claimed that such a distinctive feature is indeed what would enable SPR to go beyond the capabilities of other traditional approaches to PR. However, though parsing is perhaps the most fundamental and widely used tool of SPR, it is often used just to determine the (likelihood of) membership of the test objects to a set of classes, each of which being modeled by a different syntactic model. Any possible structural byproduct of the parsing process is therefore discarded, thus considerably wasting the high potential of SPR.

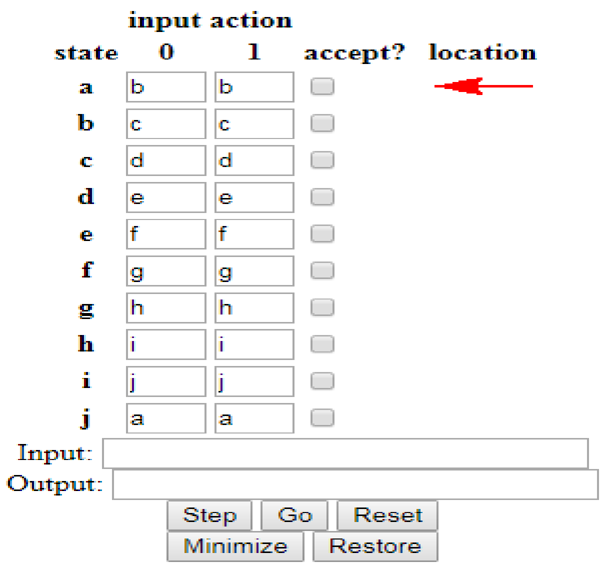
Simulator to identify the pattern The outline of our approach is the following: 1. The system to be inferred is tested and samples of its functioning are generated. Some of the samples are chosen as training data and some as testing data. 2. The number of states in the FSM is received as an input. 3. The search algorithm is applied and a FSM M is outputted. 4. M is evaluated using the given training data and/or testing data. If M describes the given input–output data sufficiently well, it is considered as result. Otherwise the search process with other parameters or training data will be repeated. 5. If required, post-processing (e.g., minimization, reduction of unreachable states) is applied.



Consider the following DFA that accepts the language L={w €(a,b)*} where w contains no more than one b.
We could view M as a specification for the following program :Until accept or reject do:

```
S: s=get-next-symbol.
    If s=end of file then accept
    Else if s=a then goto S
    Else if s=b then goto T
T: s=get-next-symbol.
    If s=end of file then accept
    Else if s=a then goto T
    Else if s=b then reject
```

The various information is stored in the database can be retrieved and matched with the given sequence effectively**.** Most common form of pattern matching involves strings of characters. In many programming languages, a particular syntax of strings is used to represent regular expressions, which are patterns describing string characters.



	input action			
state	0	1	accept?	location
a	b	b	☐	←
b	c	c	☐	
c	d	d	☐	
d	e	e	☐	
e	f	f	☐	
f	g	g	☐	
g	h	h	☐	
h	i	i	☐	
i	j	j	☐	
j	a	a	☐	

Input:
Output:

References

1. F. Aarts, P. Fiterau-Brostean, H. Kuppens, and F. W. Vaandrager. Learning Register Automata with Fresh Value Generation. In Theoretical Aspects of Computing - ICTAC, volume 9399 of LNCS, pages 165–183, 2015.
2. F. Aarts and F. Vaandrager. Learning I/O Automata. In CONCUR, pages 71–85. Springer, 2010.
3. H. I. Akram, C. de la Higuera, H. Xiao, and C. Eckert. Grammatical inference algorithms in matlab. In ICGI'10, Proceedings, pages 262–266. Springer, 2010. 17
4. A. V. Aleksandrov, S. V. Kazakov, A. A. Sergushichev, F. N. Tsarev, and A. A. Shalyto. The use of evolutionary programming based on training examples for the generation of finite state machines for controlling objects with complex behavior. J. Comput. Sys. Sc. Int., 52(3):410–425, 2013.
5. R. Alur, M. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa. Synthesizing Finite-state Protocols from Scenarios and Requirements. In HVC, volume 8855 of LNCS. Springer, 2014.
6. G. Ammons, R. Bod´ık, and J. R. Larus. Mining specifications. In Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM.
7. D. Angluin. Learning regular sets from queries and counterexamples. Inf. Comput., 75(2):87–106, 1987.
8. T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In M. Cerioli, editor, Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings, volume 3442 of Lecture Notes in Computer Science, pages 175–189. Springer, 2005.
9. A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. IEEE Trans. Comput., 21(6):592–597, June 1972.
10. I. P. Buzhinsky, V. I. Ulyantsev, D. S. Chivilikhin, and A. A. Shalyto. Inducing finite state machines from training samples using ant colony optimization. J. Comput. Sys. Sc. Int., 53(2):256–266, 2014.
11. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Learning Extended Finite State Machines. In SEFM 2014, Proceedings, pages 250–264, 2014.
12. T. S. Chow. Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng., 4(3):178–187, May 1978.
13. M. A. Colon, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In ´ Computer Aided Verification, CAV, pages 420–432. Springer, 2003.
14. F. Coste and J. Nicolas. ICGI-98, Proceedings, chapter How considering incompatible state mergings may reduce the DFA induction search tree, pages 199–210. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.
15. C. de la Higuera. Grammatical Inference: Learning Automata and Grammars. CUP, 2010.
16. R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko. Fsm-based conformance testing methods: A survey annotated with experimental evaluation. Inf. Softw. Technol., 52(12):1286–1297, Dec. 2010.
17. P. Dupont. Incremental regular inference. In ICGI-96, Proceedings, pages 222–237, 1996.
18. E. M. Gold. Language identification in the limit. Information and Control, 10(5):447–474, 1967.
19. E. M. Gold. Complexity of automaton identification from given data. Information and Control, 37(3):302–320, 1978.
20. O. Grinchtein and M. Leucker. Learning Finite-State Machines from Inexperienced Teachers. In Y. Sakakibara, S. Kobayashi, K. Sato, T. Nishino, and E. Tomita, editors, Grammatical Inference: Algorithms and Applications, 8th International Colloquium, ICGI 2006, Tokyo, Japan, September 20-22, 2006, Proceedings, volume 4201 of Lecture Notes in Computer Science, pages 344–345. Springer, 2006.
21. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In 38th POPL, pages 317–330, 2011.
22. S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, pages 281–292, New York, NY, USA, 2008. ACM.
23. A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In Computer Aided Verification, CAV, pages 634–640. Springer, 2009.