USING SYNCHRONIZING HEURISTIC FOR GENERATING UIO SEQUENCES

Student(s)
Elif Şevval Köroğlu
Selami Doğan Akansu
İlayda Tukuş
Yavuz Güleşen

Faculty Member(s)
Hüsnü Yenigün
Kamer Kaya

Sabancı Üniversitesi

PURE
PROGRAM FOR UNDERGRADUATE RESEARCH

ABSTRACT

A finite state machine or finite automaton is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. In this work, we carried out a research on automata and finite state machines, which are used as formal notations to model the behavior of systems at an abstract level. Based on the analysis of these formal models, one can derive tests for the implementations of the systems.

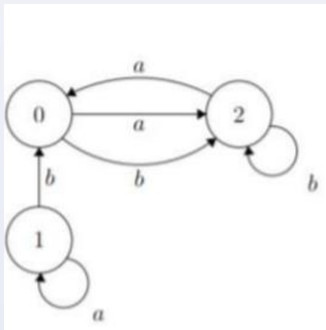


Figure 1: An automaton A_0

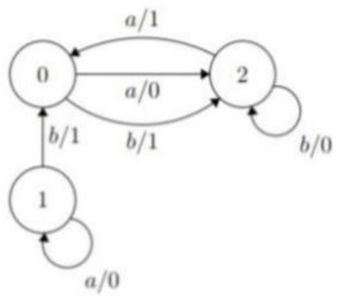


Figure 2: An FSM M_0

The FSM can change from one state to another in response to some external inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition [2].

In accordance with the general classification, the following formal definition is found:

An automaton and a FSM can be visualized as directed graph which consists on nodes and directed edges. In this visualizing system, states correspond to the nodes and the transition corresponds to the edges of the graph. For an automaton the edges of the graph are labeled by input symbols, whereas for an FSM the edges are labeled by an input and an output symbol.

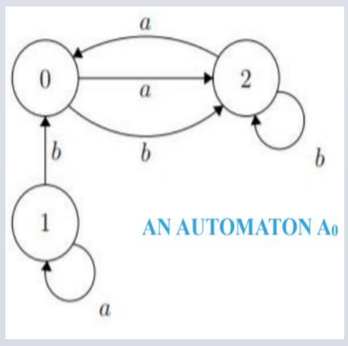
OBJECTIVES

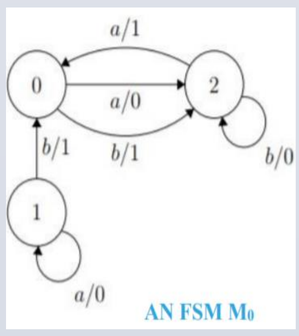
1- Our code creates random finite state machines. We try to obtain specific states's UIO. Therefore we changed active states.

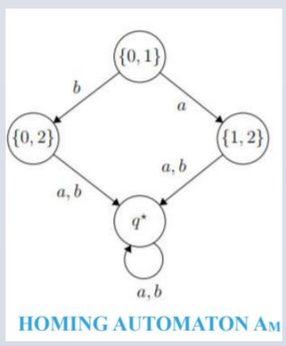
2- If there are more than one states which go another state with a same output, we changed either output or next state. In this way, we obtain an FSM where no two states merge without being distinguished.

3-We did some tests and we transfered the result data to excel format. Then we compared the Greedy and SynchroP.

PROJECT DETAILS I







All the algorithms are implemented in C++ and the automata used for these experiments are randomly generated. The purpose of this paper to find UIO sequence for FSMs which is a hard application since its function grows exponentially. Instead of using regular method, we turned the FSMs into homing automatons and modify the original code we had. In this modification process, we changed random generation part. After random generation, to find UIO for each state, we traced over all the transition and we didn't let for any 2 states s and s' ∈ S and i ∈ X, δ(s, i) = δ(s' , i) and λ(s, i) = λ(s' , i) at the same time. Because if any 2 states merge with a single input and producing same output, there is no way to separate them afterwards. After this step, we generated a special UIO automaton for this FSM. Before applying existing synchronizing sequences (Greedy and SynchroP) from [1] for UIO automaton we changed the active state set from S to special set for our algorithm. To find a UIO for state s ∈ S, we put each state of homing automaton which are the pairs of original FSM to active (to be merged) set if they consist state s. After adding all pairs containing s, we we add q* to the active set. With the existing synchronizing sequence heuristics, we tried to merge every element in active state set and as expected they all merged in q*. The synchronizing sequence that we obtained for UA is also UIO sequence for FSM. Then we compare the results of these heuristics.

PROJECT DETAILS II

For an FSM M = (S, X, Y, δ, λ), a UIO (Unique Input Output) Sequence of M is an input sequence $\bar{H} \in X^*$ such that for a state s ∈ S, $\lambda(s, \bar{H}) \neq \lambda(s', \bar{H})$ for any state s' ∈ S. For FSM $M0$ given in Figure 2, a is an UIO sequence for state 2.

Theorem. Let M = (S, X, Y, δ, λ) be an FSM and AM = (SA , X, δA) be the HA of M. An input sequence $\bar{x} \in X^*$ is an HS for M iff $\bar{x}$ is an SS for AM . Proof. If $\bar{x}$ is an HS of M, for any two states s i and s j in M, $\lambda(s_i, \bar{x}) \neq \lambda(s_j, \bar{x})$ or $\delta(s_i, \bar{x}) = \delta(s_j, \bar{x})$. Since $\delta A(\{s_i, s_j\}, \bar{x}) = q^*$ for any $\{s_i, s_j\} \in SA$. For q^* , $\delta A(q^*, \bar{x}) = q^*$. Hence $\bar{x}$ is an SS for AM .

$$\delta_A(q,x) = \begin{cases} \{\delta(s,x), \delta(s',x)\} & , (q = \{s,s'\}) \wedge (\delta(s,x) \neq \delta(s',x)) \wedge (\lambda(s,x) = \lambda(s',x)) \\ q^* & , (q = \{s,s'\}) \wedge (\delta(s,x) = \delta(s',x)) \\ q^* & , (q = \{s,s'\}) \wedge (\lambda(s,x) \neq \lambda(s',x)) \\ q^* & , (q = q^*) \end{cases}$$

```
//Creating 2 matrices for next states and outputs acording to inputs
unsigned short ** outputs = new unsigned short*[P];
unsigned short ** nextStates = new unsigned short*[P];
//Randomly filling these matrices created above
for (int i = 0; i < P; i++)
{
  nextStates[i] = new unsigned short[N];
  outputs[i] = new unsigned short[N];
  for(int j = 0; j < N; j++)
  {
    nextStates[i][j] = ((unsigned short)rand_r(&seed)) % N;
    outputs[i][j] = ((unsigned short)rand_r(&seed)) % O;
  }
}
```

```
UIO SEQUENCE FOR STATE 2
------------------------
Greedy path: a a b a a b b
Greedy path length: 7
Greedy Time: 1190430 microseconds

SynchroP path: b b b b b
SynchroP path length: 5
SynchroP Time: 1016608 microsseconds
UIO SEQUENCE FOR STATE 2
------------------------
Greedy path: a b a a b
Greedy path length: 5
Greedy Time: 40366 microseconds

SynchroP path: a b a b a
SynchroP path length: 5
SynchroP Time: 39375 microsseconds
```

CONCLUSION

We saw that the time elapsed after Greedy and SynchroP are similar. Normally SynchroP is much slower but because of the generation of UA and BFS phase of these heuristics, we thought that this time difference is normal. As expected, SynchroP is finding much shorter sequences compared to Greedy. As a future work we can try to find a way to construct a better designed UA without the necessity of the process after random generation.

states / inputs/ outputs/ selected state	Average Greedy Length	Average Greedy Time	Unit	Average syncroP Length	Average syncroP Time	Unit
16 2 2 2	4.29	569.15	microseconds	3.62	613.23	microseconds
16 2 4 2	2.5	460.47	microseconds	2.14	500.5	microseconds
16 2 8 2	1.95	387.2	microseconds	1.69	404.52	microseconds
16 2 16 2	1.56	344.39	microseconds	1.38	359.2	microseconds
16 4 2 2	3.81	658.16	microseconds	3.2	924.17	microseconds
16 4 4 2	2.43	485.95	microseconds	2.18	704.64	microseconds
16 4 8 2	1.92	428.62	microseconds	1.52	608.79	microseconds
16 8 2 2	3.71	853.54	microseconds	3.07	1832.83	microseconds
16 8 4 2	2.45	477.64	microseconds	1.95	1197.25	microseconds
16 8 8 2	1.92	453.17	microseconds	1.48	1303.45	microseconds
16 16 2 2	3.68	842.13	microseconds	2.87	3172.39	microseconds
32 2 2 2	5.31	8330.78	microseconds	4.44	8144.05	microseconds
32 2 4 2	3.07	7372.27	microseconds	2.74	7332.33	microseconds
32 2 8 2	2.37	6894.45	microseconds	2.06	6821.45	microseconds
32 2 16 2	1.88	6319.45	microseconds	1.73	6119.84	microseconds
32 4 2 2	4.89	13356.25	microseconds	3.92	15981.6	microseconds
32 4 4 2	3.02	9087.85	microseconds	2.68	11169.85	microseconds
32 4 8 2	2.36	8671.44	microseconds	1.99	10236.37	microseconds
32 4 16 2	1.88	6826.35	microseconds	1.57	8659.96	microseconds
32 8 2 2	4.8	16286.1	microseconds	3.67	28949.67	microseconds
32 8 4 2	3.04	12805.46	microseconds	2.35	23011.96	microseconds
32 8 8 2	2.36	8754.29	microseconds	1.89	17470.88	microseconds
32 16 2 2	4.77	19377.3	microseconds	3.67	54976.53	microseconds
64 2 2 2	6.24	169504.07	microseconds	5.32	134042.78	microseconds
64 2 4 2	3.61	150017.15	microseconds	3.25	115397.1	microseconds
64 2 8 2	2.64	127948.65	microseconds	2.24	95061.71	microseconds
64 2 16 2	2.18	125034.63	microseconds	1.98	92763.71	microseconds
64 4 4 2	3.56	172133.45	microseconds	2.99	161868.13	microseconds
64 4 8 2	2.63	167500.11	microseconds	2.36	149205.94	microseconds
64 8 2 2	5.69	229477.32	microseconds	4.63	334235.89	microseconds
64 8 4 2	3.54	206652.6	microseconds	2.93	294254.47	microseconds
64 8 8 2	2.62	186541.81	microseconds	2.13	238226.45	microseconds
64 16 2 2	5.67	267837.5	microseconds	4.29	609915.31	microseconds

REFERENCES

[1] D. Eppstein. Reset sequences for monotonic automata. SIAM J. Comput., 19(3):500–510, 1990.

[2] Z. Kohavi. Switching and Finite Automata Theory. McGraw–Hill, New York, 1978.

[3] B. Cirisci, M. Emek, E. Sorguc, Using Synchronizing Heuristics to Construct Homing Sequences, MODELSWARD 2019