LBT for Procedural and Reactive Systems

Part 2: Reactive Systems – Basic Theory

Karl Meinke,
karlm@kth.se

School of Computer Science and Communication
KTH Stockholm
0. Overview of the Lecture

1. Learning Based Testing for Reactive Systems
2. DFA learning with Angluin’s L* algorithm
3. Complexity of learning DFA

based on:
2. Learning-Based Testing (LBT)

“aka. Model based testing without a model”
3. Framework for Study: 

_Reactively Systems_

Generally *control-oriented* testing

1. Requirements language = propositional linear temporal logic (PLTL)
2. Model = FSM, Moore machine
3. Model checker = BDD/SAT-based checkers
4. Learning = regular inference algorithms
DFA (Moore) Representation

DFA $A = (Q, \Sigma, q_0, F \subseteq Q, \delta : Q \times \Sigma \rightarrow Q)$

System under Learning (SUL)

$L(A) = \text{regular language accepted by } A$
DFA Learning with Observation Tables

- $P_A \subseteq \Sigma^*$ is a finite prefix-closed set of prefixes
- $S_A \subseteq \Sigma^*$ is a finite suffix-closed set of suffixes

- $T_A : P_A \cup (P_A \cdot \Sigma) \times S_A \rightarrow \{1, 0, ?\}$ is the observation table

- Write $T_A(p)$ for a row $T_A(p,s_1), \ldots, T_A(p,s_n)$
OBSERVATION
TABLE

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>?</td>
</tr>
</tbody>
</table>

suffixes

prefixes
Basic Principle of DFA learning

- **Accessor strings** – prefixes \( p \) that reach each distinct state
- **Distinguishing strings** – suffixes \( s \) that separate distinct states

**INFERENCe PRINCIPLE**

If \( T_A(p, s) \neq T_A(p', s) \) then \( p \) and \( p' \) cannot reach the same state in \( A \);

\( s \) is a *distinguishing string* for \( p, p' \)
Closed & Consistent Tables

$\mathcal{T}_A$ is \textit{closed} iff, for each $p \in \mathcal{P}_A \cdot \Sigma$ there exists $p' \in \mathcal{P}_A$ st.

\[ \mathcal{T}_A(p) = \mathcal{T}_A(p') \]

$\mathcal{T}_A$ is \textit{consistent} iff, for each $p, p' \in \mathcal{P}_A$ if

\[ \mathcal{T}_A(p) = \mathcal{T}_A(p') \]

then for all $a \in \Sigma$,

\[ \mathcal{T}_A(p.a) = \mathcal{T}_A(p'.a) \]
Algebraic Properties

• Being closed is an *algebraic closure condition*
• Being consistent is an *algebraic congruence condition*
• The automaton construction is a *quotient algebra construction*

• Learning DFA as *string rewriting systems*
• K. Meinke, *CGE: a Sequential Learning Algorithm for Mealy Automata*, in Proc. ICGI 2010
Equivalence Oracles

• Termination requires an *equivalence oracle*
• If $A$ and $SUL$ are *behaviourally equivalent*, i.e. $L(A) = L(SUL)$ then
  $\text{equivOracle}(A, SUL) = \text{true}$
• Otherwise
  $\text{equivOracle}(A, SUL) = \nu \in \Sigma^*$ where $A(\nu) \neq SUL(\nu)$
• LBT uses *stochastic equivalence checking*
Complexity Observations

- Stochastic equivalence checking is more powerful than random test cases – Why?


- Using stochastic equivalence checking we can **PAC learn DFA in polynomial time** (c.f. Kearns and Vazirani 1994).
DFA function LStar(DFA: SUL)

\[ P_A \subseteq \Sigma^* \]

\[ P_A \cdot \Sigma \subseteq \Sigma^* \]

\[ S_A \subseteq \Sigma^* \]

\[ T_A : P_A \cup (P_A \cdot \Sigma) \times S_A \rightarrow \{\text{accept, reject, ?}\} \] // table

begin

\[ A = \text{getInitialHypothesis}() \]

while(equivOracle(A, SUL)! = true)) do

\[ A = \text{getNextHypothesis(equivOracle(A, SUL))} \]

return A

end
DFA function
getNextHypothesis(counterExample ∈ Σ*)

begin

$P_A = P_A \cup \text{PrefixClosure}\{\text{counterExample}\}$

$P_A \cdot \Sigma = P_A \times \Sigma - P_A$

$S_A = S_A \cup \text{SuffixClosure}\{\text{counterExample}\}$

// fill in any new table entries here ...

while $(P_A, S_A, T_A)$ is not closed or consistent do

    if !consistent($P_A, S_A, T_A$) makeConsistent()

    if !closed($P_A, S_A, T_A$) makeClosed()

end
DFA function getInitialHypothesis()
begin
  \( P_A = \emptyset \)  // emptyset
  \( P_A \cdot \Sigma = \emptyset \)
  \( S_A = \emptyset \)
  return getNextHypothesis(\varepsilon)
end
DFA function
makeConsistent()
begin
  find $p, p' \in P_A$, $a \in \Sigma$, $s \in S_A$ st.
  $T_{A}(p) = T_{A}(p')$ and
  $T_{A}(p.a,s) \neq T_{A}(p'.a,s)$
  add $a.s$ to $S_A$ // suffix extension
  extend $T_A$ to $P_A \cup (P_A.\Sigma) \times S_A$ using
  active membership queries
end
DFA function
makeClosed()
begin

find p ∈ PA, a ∈ Σ st.

TA(p.a) != TA(p') for all p' ∈ PA

PA = PA ∪ {p.a} // prefix extension

PA.Σ = PA.Σ ∪ {p}×Σ

extend TA to PA ∪ (PA.Σ) X SA using membership queries

end
DFA function DFASynthesis()
begin
Q = \{u : u \in P_A, \forall v < u, T_A(u) \neq T_A(v) \}\nq_0 = \varepsilon
F = \{u : u \in Q, T_A(u, \varepsilon) = 1\}
foreach u \in Q do
    foreach a \in \Sigma do
        \delta(u, a) = v \in Q st. T_A(u.a) = T_A(v)
return A = (Q, \Sigma, q_0, F, \delta)
end
Let’s L* learn this DFA
<table>
<thead>
<tr>
<th></th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Closed: no  
Consistent: yes

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

Closed: yes  
Consistent: yes
equivOracle($H_0$, SUL)  = 110 = counterExample

<table>
<thead>
<tr>
<th></th>
<th>$\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>

Closed: yes  
Consistent: no

Not consistent because

$T_A(\epsilon) = T_A(11) = 1$

but

$T_A(1) \neq T_A(111)$ since $0 \neq 1$

Also

$T_A(0) \neq T_A(110)$ since $0 \neq 1$
equivOracle\( (H_0, \text{ SUL}) \) = 110 = counterExample

<table>
<thead>
<tr>
<th>( \varepsilon )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varepsilon )</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Closed: yes  
Consistent: no

<table>
<thead>
<tr>
<th>( \varepsilon )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varepsilon )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Closed: yes  
Consistent: yes

\( H_1 = \text{FINISHED!} \)
Theoretical Complexity

Membership queries = $O(m|\Sigma||Q|^2)$

where $m$ is the maximum length of any counterexample (Angluin 87).

Assume oracle returns shortest counterexample then $Q \geq m$, so

Membership queries = $O(|\Sigma||Q|^3)$
The alphabet size is fixed to 10.
5. Conclusions

- L* is pedagogically easy to learn
- some of its principles are universal
- looks promising on paper
- emphasis on “complete learning”
- LBT needs “incremental learning”

Open Questions

- How complex are real SUTs?
- Can try to benchmark other learning algorithms in same framework