LearnLib Tutorial

Malte Isberner\textsuperscript{1}, Bernhard Steffen\textsuperscript{1}, Falk Howar\textsuperscript{2}

\textsuperscript{1}: TU Dortmund University, Chair for Programming Systems, Germany
\textsuperscript{2}: IPSSE / TU Clausthal, Germany

September 22, 2015
RV 2015, Vienna, AT
Organization

Malte Isberner
TU Dortmund University, Germany

Bernhard Steffen
TU Dortmund University, Germany

Falk Howar
IPSSE / TU Clausthal, Germany
Outline

1. Introduction
2. Active Automata Learning
3. Automata Learning in Practice
4. The TTT Algorithm
5. LearnLib
Outline

1. Introduction
2. Active Automata Learning
3. Automata Learning in Practice
4. The TTT Algorithm
5. LearnLib
Motivation

Aim of this tutorial ...

- introduce (some of) the theory behind active automata learning
- discuss what are the key challenges when using active automata learning in practice
- motivate the TTT algorithm, which is the only algorithm to properly address the problem of long counterexamples
- give an overview of LearnLib
Outline

1. Introduction
2. Active Automata Learning
3. Automata Learning in Practice
4. The TTT Algorithm
5. LearnLib
High-Level Overview

Learner

Target system

High-Level Overview

Introduction  Active Automata Learning  Automata Learning in Practice  The TTT Algorithm  LearnLib
test case (API calls, …)
High-Level Overview

test case (API calls, ...)

output (trace)

The learner interacts with the target system to generate test cases and observe their outputs.
High-Level Overview

- **test case (API calls, ...)**
- **output (trace)**
- **equivalent?**
High-Level Overview

test case (API calls, \ldots)

output (trace)

Yes: done

equivalent?
High-Level Overview

test case (API calls, \ldots)

\begin{itemize}
  \item Yes: done
  \item No: counterexample
\end{itemize}

\( w \) exposing difference between target system and \( \mathcal{H} \)

equivalent?
Active Automata Learning: Construct (behavioral) models of (black-box) systems via testing
- Typically: finite-state models (DFAs or Mealy machines)
- “Inverse” of model-based testing: test-based modeling!
Active Automata Learning

- **Active Automata Learning**: Construct (behavioral) models of (black-box) systems via testing
  - Typically: finite-state models (DFAs or Mealy machines)
  - “Inverse” of model-based testing: test-based modeling!
- Interaction with target systems via queries
  - Membership Queries (MQs): what is the response to a sequence of inputs?
  - Equivalence Queries (EQs): does the hypothesis correctly and completely model the behavior of the SUL (system under learning)?
Active Automata Learning: Construct (behavioral) models of (black-box) systems via testing

- Typically: finite-state models (DFAs or Mealy machines)
- “Inverse” of model-based testing: test-based modeling!

Interaction with target systems via queries

- Membership Queries (MQs): what is the response to a sequence of inputs?
- Equivalence Queries (EQs): does the hypothesis correctly and completely model the behavior of the SUL (system under learning)?

So what is this good for?
(small) learned models imposed
major test suite optimizations
Another Application: Learning Models of Embedded Control Software

Smenk, Moerman, Jansen, Vaandrager: Applying Automata Learning to Embedded Control Software (ICFEM 2015)
Another Application: Learning Models of

Smenk, Moerman, Jansen, Vaandrager: Applying Automata Learning to Embedded Control Software (ICFEM 2015)
Lots and Lots of Applications, Actually ...

- Black-box model checking [Peled et al., FORTE’99]
- Test-case generation [Hagerer et al., FASE’02]
- Assume-guarantee style compositional verification [Cobleigh et al., TACAS’03]
- Interface synthesis [Alur et al., POPL’05; Giannakopoulou et al., SAS’12]
- Botnet analysis [Cho et al., CCS’10]
- Connector synthesis [Issarny et al., 2009]
- GUI testing [Choi et al., OOPSLA’13]
Restrictions: System Requirements

What are the restrictions for applying active automata learning?

- **Finite input alphabet** $\Sigma$
- **Finite-state** ("regular") control structure
- **Determinism**
- **Ability to reset**
  - $\Rightarrow$ independence of subsequent membership queries
What are the restrictions for applying active automata learning?

- **Finite input alphabet** $\Sigma$
- **Finite-state** ("regular") control structure
- **Determinism**
- **Ability to reset**
  $\Rightarrow$ independence of subsequent membership queries

Most of the above: matter of choosing the right *abstraction!*
Restrictions: Output Models

What kind of models can be learned?

- Mealy machines [Niese 2003; Shahbaz&Groz 2009]
  - Based on this: restricted forms I/O automata
    [Aarts&Vaandrager 2010]
Restrictions: Output Models

What kind of models can be learned?

- Mealy machines [Niese 2003; Shahbaz&Groz 2009]
  - Based on this: restricted forms I/O automata [Aarts&Vaandrager 2010]
- Classical formulation: DFAs [Angluin 1987]
  - Allows for simpler presentation
  - Adaptation to Mealy machines pretty straightforward
Technical Realization: Setup

Assumption: the behavior of the target system is modeled by a DFA $A$

- **Membership Queries** ask if a word $w \in \Sigma^*$ is accepted by $A$
- **Notation:** $\lambda(w) = 1$ iff $w$ is accepted by $A$ (otherwise 0) ($\lambda$: “output function”)
  
  $\Rightarrow$ Membership query $\hat{=} \lambda$-evaluation
Main challenge: reasoning about structure (states and transitions) of unknown DFA $\mathcal{A}$
Technical Realization: Main Idea

Main challenge: reasoning about structure (states and transitions) of unknown DFA $\mathcal{A}$

- Represent states of $\mathcal{A}$ by words $u \in \Sigma^*$ reaching them
  - Notation: $\mathcal{A}[u] \doteqdot$ state in $\mathcal{A}$ reached by $u$
  - Determinism and independence of membership queries ensure well-definedness
- Maintain set $\mathcal{U}$ of “short prefixes”
Technical Realization: Main Idea

Main challenge: reasoning about structure (states and transitions) of unknown DFA $\mathcal{A}$

- Represent states of $\mathcal{A}$ by words $u \in \Sigma^*$ reaching them
  - Notation: $\mathcal{A}[u] \doteq$ state in $\mathcal{A}$ reached by $u$
  - Determinism and independence of membership queries ensure well-definedness
- Maintain set $\mathcal{U}$ of “short prefixes”
- For $a \in \Sigma$: $a$-successor of $\mathcal{A}[u]$ is $\mathcal{A}[ua]$
  - Allows to reference transitions
  - Set $\mathcal{U} \cdot \Sigma$ of “long prefixes”
Main challenge: reasoning about structure (states and transitions) of unknown DFA $\mathcal{A}$

- Represent states of $\mathcal{A}$ by words $u \in \Sigma^*$ reaching them
  - Notation: $\mathcal{A}[u] \doteq$ state in $\mathcal{A}$ reached by $u$
  - Determinism and independence of membership queries ensure well-definedness
- Maintain set $\mathcal{U}$ of “short prefixes”
- For $a \in \Sigma$: $a$-successor of $\mathcal{A}[u]$ is $\mathcal{A}[ua]$
  - Allows to reference transitions
  - Set $\mathcal{U} \cdot \Sigma$ of “long prefixes”
- States $\mathcal{A}[u]$ and $\mathcal{A}[u']$ are (observably) different iff
  $\lambda(u \cdot v) \neq \lambda(u' \cdot v)$ for some $v \in \Sigma^*$
  - $v$ called distinguishing suffix (for $u$ and $u'$)
  - Challenge: finding distinguishing suffixes (cannot search through the whole of $\Sigma^*$)
Realization: Observation Table

Data structure of $L^*$ (and similar algorithms): Observation Table

<table>
<thead>
<tr>
<th></th>
<th>$\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>1</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
</tr>
<tr>
<td>$bb$</td>
<td>0</td>
</tr>
</tbody>
</table>

*Diagram of automaton A with states $q_0$, $q_2$, and $q_1$ with transitions labeled $a$, $b$, and $a,b$.*
Data structure of $L^*$ (and similar algorithms): Observation Table

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>1</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
</tr>
<tr>
<td>$bb$</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Rows:**
  - **Upper part:** Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify states
Realization: Observation Table

Data structure of $L^*$ (and similar algorithms): Observation Table

| $\varepsilon$ | $\varepsilon$ | 1 |
| $b$ | 0 |
| $a$ | 1 |
| $ba$ | 0 |
| $bb$ | 0 |

- **Rows:**
  - **Upper part:** Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - **Lower part:** Long prefixes identify **transitions** (one-letter extensions)
Realization: Observation Table

Data structure of L* (and similar algorithms): Observation Table

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>ε</th>
<th>b</th>
<th>a</th>
<th>ba</th>
<th>bb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ba</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bb</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Rows:**
  - **Upper part:** Short prefixes \( \mathcal{U} = \{\varepsilon, b\} \) identify states
  - **Lower part:** Long prefixes identify transitions (one-letter extensions)

- **Columns:** Distinguishing suffixes
# Realization: Observation Table

Data structure of $L^*$ (and similar algorithms): Observation Table

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$b$</th>
<th>$a$</th>
<th>$ba$</th>
<th>$bb$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Rows:**
  - **Upper part:** Short prefixes ($U = \{\varepsilon, b\}$) identify **states**
  - **Lower part:** Long prefixes identify **transitions** (one-letter extensions)

- **Columns:** Distinguishing suffixes

- **Cell contents** (row $u$, col. $v$): $\lambda(u \cdot v)$
Realization: Observation Table

Data structure of $L^*$ (and similar algorithms): Observation Table

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$ba$</th>
<th>$bb$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>$1$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

- **Rows:**
  - **Upper part:** Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify states
  - **Lower part:** Long prefixes identify transitions (one-letter extensions)

- **Columns:** Distinguishing suffixes
- **Cell contents** (row $u$, col. $v$): $\lambda(u \cdot v)$

Diagram:

- States: $q_0, q_1, q_2$
- Transitions: $a, b$
- Start state: $q_0$
- Accepting states: $q_1, q_2$
Data structure of $L^*$ (and similar algorithms): Observation Table

<table>
<thead>
<tr>
<th>Row</th>
<th>$\varepsilon$</th>
<th>$b$</th>
<th>$a$</th>
<th>$ba$</th>
<th>$bb$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Rows:**
  - **Upper part:** Short prefixes ($U = \{\varepsilon, b\}$) identify states
  - **Lower part:** Long prefixes identify transitions (one-letter extensions)

- **Columns:** Distinguishing suffixes

- **Cell contents** (row $u$, col. $v$): $\lambda(u \cdot v)$
Realization: Observation Table

Data structure of $L^*$ (and similar algorithms): Observation Table

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>1</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
</tr>
<tr>
<td>$bb$</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Rows:**
  - **Upper part:** Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - **Lower part:** Long prefixes identify **transitions** (one-letter extensions)

- **Columns:** Distinguishing suffixes

- **Cell contents** (row $u$, col. $v$): $\lambda(u \cdot v)$
Realization: Observation Table

Data structure of $L^*$ (and similar algorithms): Observation Table

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$b$</th>
<th>$a$</th>
<th>$ba$</th>
<th>$bb$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$bb$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Rows:**
  - **Upper part:** Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - **Lower part:** Long prefixes identify **transitions** (one-letter extensions)

- **Columns:** Distinguishing suffixes

- **Cell contents** (row $u$, col. $v$): $\lambda(u \cdot v)$
Realization: Observation Table

Data structure of \( L^* \) (and similar algorithms): Observation Table

<table>
<thead>
<tr>
<th></th>
<th>( \varepsilon )</th>
<th>( b )</th>
<th>( a )</th>
<th>( ba )</th>
<th>( bb )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varepsilon )</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( b )</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( a )</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ba )</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( bb )</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Rows**:  
  - **Upper part**: Short prefixes \( U = \{ \varepsilon, b \} \) identify **states**  
  - **Lower part**: Long prefixes identify **transitions** (one-letter extensions)

- **Columns**: Distinguishing suffixes

- **Cell contents** (row \( u \), col. \( v \)): \( \lambda(u \cdot v) \)
Realization: Observation Table

Data structure of $L^*$ (and similar algorithms): Observation Table

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$H$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>1</td>
<td>$[\varepsilon]$</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td>$b$</td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
<td>$a$</td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
<td>$ba$</td>
</tr>
<tr>
<td>$bb$</td>
<td>0</td>
<td>$bb$</td>
</tr>
</tbody>
</table>

- **Rows:**
  - **Upper part:** Short prefixes ($U = \{\varepsilon, b\}$) identify **states**
  - **Lower part:** Long prefixes identify **transitions** (one-letter extensions)

- **Columns:** Distinguishing suffixes

- **Cell contents** (row $u$, col. $v$): $\lambda(u \cdot v)$
## Realization: Observation Table

**Data structure of \( L^* \) (and similar algorithms): Observation Table**

<table>
<thead>
<tr>
<th>( \varepsilon )</th>
<th>( \varepsilon )</th>
<th>( b )</th>
<th>( a )</th>
<th>( ba )</th>
<th>( bb )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varepsilon )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( b )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( a )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ba )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( bb )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- **Rows:**
  - **Upper part:** Short prefixes \( \mathcal{U} = \{ \varepsilon, b \} \) identify states
  - **Lower part:** Long prefixes identify transitions (one-letter extensions)

- **Columns:** Distinguishing suffixes

- **Cell contents** (row \( u \), col. \( v \)): \( \lambda(u \cdot v) \)
### Data structure of L* (and similar algorithms): Observation Table

<table>
<thead>
<tr>
<th>(\varepsilon)</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\varepsilon)</td>
<td>1</td>
</tr>
<tr>
<td>(b)</td>
<td>0</td>
</tr>
<tr>
<td>(a)</td>
<td>1</td>
</tr>
<tr>
<td>(ba)</td>
<td>0</td>
</tr>
<tr>
<td>(bb)</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Rows:**
  - **Upper part:** Short prefixes \(\mathcal{U} = \{\varepsilon, b\}\) identify **states**
  - **Lower part:** Long prefixes identify **transitions** (one-letter extensions)

- **Columns:** **Distinguishing suffixes**

- **Cell contents (row \(u\), col. \(v\)):** \(\lambda(u \cdot v)\)

---

**Diagram:**

- \(\mathcal{H}\): Transition graph for \(a, b\)
- \(\mathcal{A}\): Automaton graph with states \(q_0, q_1, q_2\) and transitions on \(a, b\)
Realization: Observation Table

Data structure of $L^*$ (and similar algorithms): Observation Table

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$b$</th>
<th>$a$</th>
<th>$ba$</th>
<th>$bb$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$bb$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Rows:**
  - **Upper part:** Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify states
  - **Lower part:** Long prefixes identify transitions (one-letter extensions)

- **Columns:** Distinguishing suffixes
- **Cell contents** (row $u$, col. $v$): $\lambda(u \cdot v)$
Refinement

<table>
<thead>
<tr>
<th>ε</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>ba</td>
<td>0</td>
</tr>
<tr>
<td>bb</td>
<td>0</td>
</tr>
</tbody>
</table>

- $\mathcal{H}$ and $\mathcal{A}$ not equivalent!
Refinement

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>$1$</td>
</tr>
<tr>
<td>$b$</td>
<td>$0$</td>
</tr>
<tr>
<td>$a$</td>
<td>$1$</td>
</tr>
<tr>
<td>$ba$</td>
<td>$0$</td>
</tr>
<tr>
<td>$bb$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

$\mathcal{H}$ and $\mathcal{A}$ not equivalent!

Reason: $q_0$ and $q_2$ in $\mathcal{A}$ have same future behavior wrt. $\varepsilon$
Refinement

<table>
<thead>
<tr>
<th></th>
<th>( \varepsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varepsilon )</td>
<td>1</td>
</tr>
<tr>
<td>( b )</td>
<td>0</td>
</tr>
<tr>
<td>( a )</td>
<td>1</td>
</tr>
<tr>
<td>( ba )</td>
<td>0</td>
</tr>
<tr>
<td>( bb )</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \mathcal{H} \]

\[ \mathcal{A} \]

- \( \mathcal{H} \) and \( \mathcal{A} \) not equivalent!
- Reason: \( q_0 \) and \( q_2 \) in \( \mathcal{A} \) have same future behavior wrt. \( \varepsilon \)
  - \( \varepsilon \) alone not sufficient for distinguishing \( q_0 \) (\( \varepsilon \)) and \( q_2 \) (\( a \))
  - additional columns needed
Refinement

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ba</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>bb</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

\[ H \]

\[ \begin{array}{c}
q_0 \\
q_2 \\
q_1
\end{array} \]

- **H** and **A** not equivalent!
- **Reason:** \( q_0 \) and \( q_2 \) in **A** have same future behavior wrt. \( \varepsilon \)
  - \( \Rightarrow \) \( \varepsilon \) alone not sufficient for distinguishing \( q_0 (\varepsilon) \) and \( q_2 (a) \)
  - \( \Rightarrow \) additional columns needed
Refinement

<table>
<thead>
<tr>
<th>ε</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>1 0</td>
</tr>
<tr>
<td>b</td>
<td>0 0</td>
</tr>
<tr>
<td>a</td>
<td>1 1</td>
</tr>
<tr>
<td>ba</td>
<td>0 0</td>
</tr>
<tr>
<td>bb</td>
<td>0 0</td>
</tr>
</tbody>
</table>

\[ \mathcal{H} \]

\[ \mathcal{A} \]

- \( \mathcal{H} \) and \( \mathcal{A} \) not equivalent!
- Reason: \( q_0 \) and \( q_2 \) in \( \mathcal{A} \) have same future behavior wrt. \( \varepsilon \)
  - \( \varepsilon \) alone not sufficient for distinguishing \( q_0 (\varepsilon) \) and \( q_2 (a) \)
  - additional columns needed
Refinement

<table>
<thead>
<tr>
<th></th>
<th>(\varepsilon)</th>
<th>(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\varepsilon)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(b)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(a)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(ba)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(bb)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \mathcal{H} \]

\[
\begin{array}{c}
[\varepsilon] \\
\downarrow \varepsilon \\
[b]
\end{array}
\]

\[ \mathcal{A} \]

\[
\begin{array}{c}
q_0 \\
\downarrow b \\
q_1
\end{array}
\]

\[
\begin{array}{c}
q_0 \\
\downarrow a \\
q_2
\end{array}
\]

\[
\begin{array}{c}
q_2 \\
\downarrow a, b \\
q_2
\end{array}
\]

- \(\mathcal{H}\) and \(\mathcal{A}\) not equivalent!
- Reason: \(q_0\) and \(q_2\) in \(\mathcal{A}\) have same future behavior wrt. \(\varepsilon\)
  \[ \Rightarrow \varepsilon \text{ alone not sufficient for distinguishing } q_0 \ (\varepsilon) \text{ and } q_2 \ (a) \]
  \[ \Rightarrow \text{additional columns needed} \]
Refinement

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$bb$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$\mathcal{H}$ and $\mathcal{A}$ not equivalent!

- Reason: $q_0$ and $q_2$ in $\mathcal{A}$ have same future behavior wrt. $\varepsilon$
  - $\varepsilon$ alone not sufficient for distinguishing $q_0$ ($\varepsilon$) and $q_2$ ($a$)
  - additional columns needed
Refinement

<table>
<thead>
<tr>
<th>$\varepsilon$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>1 0</td>
</tr>
<tr>
<td>$b$</td>
<td>0 0</td>
</tr>
<tr>
<td>$a$</td>
<td>1 1</td>
</tr>
<tr>
<td>$ba$</td>
<td>0 0</td>
</tr>
<tr>
<td>$bb$</td>
<td>0 0</td>
</tr>
<tr>
<td>$aa$</td>
<td>0 0</td>
</tr>
<tr>
<td>$ab$</td>
<td>1 1</td>
</tr>
</tbody>
</table>

- $\mathcal{H}$ and $\mathcal{A}$ not equivalent!
- Reason: $q_0$ and $q_2$ in $\mathcal{A}$ have same future behavior wrt. $\varepsilon$ 
  $\Rightarrow$ $\varepsilon$ alone not sufficient for distinguishing $q_0$ ($\varepsilon$) and $q_2$ ($a$)
  $\Rightarrow$ additional columns needed
Refinement

### Table

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$bb$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$aa$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$ab$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Automata $H$ and $A$

- $H$ and $A$ not equivalent!
- Reason: $q_0$ and $q_2$ in $A$ have same future behavior wrt. $\varepsilon$
  - $\varepsilon$ alone not sufficient for distinguishing $q_0$ ($\varepsilon$) and $q_2$ ($a$)
  - additional columns needed
Refinement

\[\begin{array}{c|cc}
& \varepsilon & b \\
\hline
\varepsilon & 1 & 0 \\
b & 0 & 0 \\
a & 1 & 1 \\
ba & 0 & 0 \\
bb & 0 & 0 \\
aa & 0 & 0 \\
ab & 1 & 1 \\
\end{array}\]

- \(\mathcal{H}\) and \(\mathcal{A}\) not equivalent!
- Reason: \(q_0\) and \(q_2\) in \(\mathcal{A}\) have same future behavior wrt. \(\varepsilon\)
  \[\Rightarrow\] \(\varepsilon\) alone not sufficient for distinguishing \(q_0\) (\(\varepsilon\)) and \(q_2\) (\(a\))
  \[\Rightarrow\] additional columns needed
Refinement: Counterexamples

Reminder: original situation

\[ H \]

\[
\begin{array}{c}
\varepsilon \\
\rightarrow \\
[\varepsilon] \\
\rightarrow b \\
[b] \\
a \\
\end{array}
\]

\[
\begin{array}{c}
a, b \\
\rightarrow \\
[\varepsilon] \\
\rightarrow b \\
[b] \\
a \\
\end{array}
\]

- Previous slide: adding new distinguishing suffix \( b \) triggered refinement
- Counterexamples can be used to obtain such a suffix
  - Counterexample: word \( w \in \Sigma^* \) s.t. \( \lambda_H(w) \neq \lambda(w) \)
    \[ \Rightarrow \text{ i.e., } H \text{ predicts wrong output for } w \]
- Example: \( w = ab. \lambda(ab) = 1, \lambda_H(ab) = 0 \)
Counterexample Decomposition

If \( w \) is a counterexample wrt. \( \mathcal{H} \), then \( w \) has a suffix \( av \) s.t. for two access sequences \( u, u' \in \mathcal{U} \) such that \( ua \) and \( u' \) reach the same state in \( \mathcal{H} \), we have

\[
\lambda(ua \cdot v) \neq \lambda(u' \cdot v).
\]

- \( u' \) in upper part of table, \( ua \) in lower part (since \( u \in \mathcal{U} \)), contents of both rows are equal
- \( \Rightarrow \) adding suffix \( v \) to table distinguishes \( ua \) and \( u' \)
- \( \Rightarrow \) \( ua \) gets moved to upper part
Counterexample Decomposition

If \( w \) is a counterexample wrt. \( \mathcal{H} \), then \( w \) has a suffix \( av \) s.t. for two access sequences \( u, u' \in \mathcal{U} \) such that \( ua \) and \( u' \) reach the same state in \( \mathcal{H} \), we have

\[
\lambda(ua \cdot v) \neq \lambda(u' \cdot v).
\]
Counterexample Decomposition

If \( w \) is a counterexample wrt. \( \mathcal{H} \), then \( w \) has a suffix \( av \) s.t. for two access sequences \( u, u' \in \mathcal{U} \) such that \( ua \) and \( u' \) reach the same state in \( \mathcal{H} \), we have

\[
\lambda(ua \cdot v) \neq \lambda(u' \cdot v).
\]
Counterexample Decomposition

If \( w \) is a counterexample wrt. \( \mathcal{H} \), then \( w \) has a suffix \( av \) s.t. for two access sequences \( u, u' \in \mathcal{U} \) such that \( ua \) and \( u' \) reach the same state in \( \mathcal{H} \), we have

\[
\lambda(ua \cdot v) \neq \lambda(u' \cdot v).
\]
If $w$ is a counterexample wrt. $\mathcal{H}$, then $w$ has a suffix $av$ s.t. for two access sequences $u, u' \in \mathcal{U}$ such that $ua$ and $u'$ reach the same state in $\mathcal{H}$, we have

$$\lambda(ua \cdot v) \neq \lambda(u' \cdot v).$$
Counterexample Decomposition

If $w$ is a counterexample wrt. $\mathcal{H}$, then $w$ has a suffix $av$ s.t. for two access sequences $u, u' \in \mathcal{U}$ such that $ua$ and $u'$ reach the same state in $\mathcal{H}$, we have

$$\lambda(ua \cdot v) \neq \lambda(u' \cdot v).$$
Counterexample Decomposition: Visualization

Counterexample Decomposition

If $w$ is a counterexample wrt. $\mathcal{H}$, then $w$ has a suffix $av$ s.t. for two access sequences $u, u' \in \mathcal{U}$ such that $ua$ and $u'$ reach the same state in $\mathcal{H}$, we have

$$\lambda(ua \cdot v) \neq \lambda(u' \cdot v).$$
Finding a Decomposition

Counterexample Decomposition

If $w$ is a counterexample wrt. $H$, then $w$ has a suffix $av$ s.t. for two access sequences $u, u' \in U$ such that $ua$ and $u'$ reach the same state in $H$, we have

$$
\lambda(ua \cdot v) \neq \lambda(u' \cdot v).
$$

- Maler&Pnueli (1995): adding all suffixes of counterexample $w$ guarantees that “right” suffix $v$ gets added
  - additional suffixes do not hurt (wrt. refinement)
  - table size ($\hat{\gamma}$ required number of queries/test cases) grows drastically
Finding a Decomposition

Counterexample Decomposition

If \( w \) is a counterexample wrt. \( \mathcal{H} \), then \( w \) has a suffix \( av \) s.t. for two access sequences \( u, u' \in U \) such that \( ua \) and \( u' \) reach the same state in \( \mathcal{H} \), we have

\[
\lambda(ua \cdot v) \neq \lambda(u' \cdot v).
\]

- Maler\&Pnueli (1995): adding all suffixes of counterexample \( w \) guarantees that “right” suffix \( v \) gets added
  - additional suffixes do not hurt (wrt. refinement)
  - table size (\( \hat{=} \) required number of queries/test cases) grows drastically
- Rivest\&Schapire (1993): use binary search to determine \( v \)
Binary Search

Notation/terminology:

- (Reminder) $\mathcal{H}[w] \doteq$ state in $\mathcal{H}$ reached by $w = w_1 \ldots w_m$

- Access sequence of $q \in Q^\mathcal{H}$: unique element $u \in U$ representing state $q$ (furthermore: $\mathcal{H}[u] = q$)

- $[w]_\mathcal{H} \doteq$ access sequence of state $\mathcal{H}[w]$

- $\pi_\mathcal{H}(w, i) = df \ [w_1 \ldots w_i]_\mathcal{H} w_{i+1} \ldots w_m \ (i \in \{0, \ldots, m\})$: prefix transformation
  - $\pi_\mathcal{H}(w, 0) = w \Rightarrow \lambda(\pi_\mathcal{H}(w, 0)) \neq \lambda_\mathcal{H}(\pi_\mathcal{H}(w, 0))$ ($w$ is a counterexample)
  - $\pi_\mathcal{H}(w, m) = [w]_\mathcal{H} \in U \Rightarrow \lambda(\pi_\mathcal{H}(w, m)) = \lambda_\mathcal{H}(\pi_\mathcal{H}(w, m))$ ($\lambda(\pi_\mathcal{H}(w, m) \cdot \varepsilon)$ determines acceptance of state represented by $\pi_\mathcal{H}(w, m) \in U$)
- $\pi_H(w, i) = \text{def } [w_1 \ldots w_i]_H w_{i+1} \ldots w_m \ (i \in \{0, \ldots, m\})$:
  - prefix transformation
  - $\pi_H(w, 0) = w \Rightarrow \lambda(\pi_H(w, 0)) \neq \lambda_H(\pi_H(w, 0))$
    ($w$ is a counterexample)
  - $\pi_H(w, m) = [w]_H \in U \Rightarrow \lambda(\pi_H(w, m)) = \lambda_H(\pi_H(w, m))$
    ($\lambda(\pi_H(w, m) \cdot \varepsilon)$ determines acceptance of state represented by $\pi_H(w, m) \in U$)

- Obviously: $\lambda_H$ invariant under prefix transformations (i.e.,
  $\lambda_H(\pi_H(w, i)) = \lambda_H(w)$ for all $i$)

- Breakpoint: $i \in \{0, \ldots, m - 1\}$ s.t.
  $\lambda(\pi_H(w, i)) \neq \lambda(\pi_H(w, i + 1))$
  - exists if $w$ is a counterexample
Breakpoints

- **Breakpoint:** \( i \in \{0, \ldots, m - 1\} \) s.t.
  \[ \lambda(\pi_H(w, i)) \neq \lambda(\pi_H(w, i + 1)) \]

  \[
  \lambda(\lfloor w_1 \ldots w_i \rfloor_H w_{i+1} \cdot w_{i+2} \ldots w_m) \\
  \neq \lambda(\lfloor w_1 \ldots w_{i+1} \rfloor_H \cdot w_{i+2} \ldots w_m)
  \]
Breakpoints

■ **Breakpoint:** \( i \in \{0, \ldots, m - 1\} \) s.t.
\[
\lambda(\pi_{\mathcal{H}}(w, i)) \neq \lambda(\pi_{\mathcal{H}}(w, i + 1))
\]

\[
\lambda(\underbrace{[w_1 \ldots w_i]}_{u} \underbrace{\mathcal{H}}_{a} \underbrace{w_{i+1}}_{v} \cdot \underbrace{w_{i+2} \ldots w_m}_{v'})
\neq
\lambda(\underbrace{[w_1 \ldots w_{i+1}]}_{u'} \underbrace{\mathcal{H}}_{a} \cdot \underbrace{w_{i+2} \ldots w_m}_{v})
\]

■ **with** \( u = df \ [w_1 \ldots w_i] \mathcal{H}, \ a = df \ w_{i+1}, \ v = df \ w_{i+2} \ldots w_m; \)

\( \Rightarrow \) decomposition theorem
Breakpoints

- **Breakpoint:** $i \in \{0, \ldots, m - 1\}$ s.t.
  \[ \lambda(\pi_H(w, i)) \neq \lambda(\pi_H(w, i + 1)) \]

  \[
  \lambda(\left[\begin{array}{c} w_1 \ldots w_i \\ H \end{array}\right] H w_{i+1} \cdot \left[\begin{array}{c} w_{i+2} \ldots w_m \\ v \end{array}\right]) \\
  \neq \lambda(\left[\begin{array}{c} w_1 \ldots w_{i+1} \\ u' \end{array}\right] H \cdot \left[\begin{array}{c} w_{i+2} \ldots w_m \\ v \end{array}\right])
  \]

- **with** $u =_{df} \left[\begin{array}{c} w_1 \ldots w_i \\ H \end{array}\right] H, a =_{df} w_{i+1}, v =_{df} w_{i+2} \ldots w_m$;
  \[ \Rightarrow \text{decomposition theorem} \]

**Counterexample Decomposition**

If $w$ is a counterexample wrt. $H$, then $w$ has a suffix $av$ s.t. for two access sequences $u, u' \in U$ such that $ua$ and $u'$ reach the same state in $H$, we have

\[ \lambda(ua \cdot v) \neq \lambda(u' \cdot v). \]
The Cost of Learning

Common measure: query complexity (\(\hat{\text{required number of test cases}}\))
The Cost of Learning

Common measure: query complexity (\( \hat{=} \) required number of test cases)

- Parameters:
  - \( n \hat{=} \) number of states of \( A \)
  - \( k \hat{=} \) size of \( \Sigma \)
  - \( m \hat{=} \) length of longest counterexamples

How many counterexamples (equivalence queries)?
- Every counterexample leads to at least one new state in \( H \)
- \( n - 1 \) counterexamples are sufficient

Query complexity dominated by size of observation table (1 test case per cell)
- \( kn + 1 \) rows
- \( 1 \) suffix added per counterexample \( \Rightarrow \) at most \( n \) columns
- \( \Rightarrow \) size of table in \( O(\text{k}^n) \)
The Cost of Learning

Common measure: query complexity (\( \hat{=} \) required number of test cases)

- Parameters:
  - \( n \hat{=} \) number of states of \( A \)
  - \( k \hat{=} \) size of \( \Sigma \)
  - \( m \hat{=} \) length of longest counterexamples

- How many counterexamples (equivalence queries)?
  - Every counterexample leads to at least one new state in \( \mathcal{H} \)
  - \( n - 1 \) counterexamples are sufficient
The Cost of Learning

Common measure: query complexity (\(\hat{n}\) required number of test cases)

- **Parameters:**
  - \(n\) \(\hat{=}\) number of states of \(A\)
  - \(k\) \(\hat{=}\) size of \(\Sigma\)
  - \(m\) \(\hat{=}\) length of longest counterexamples

- **How many counterexamples (equivalence queries)?**
  - Every counterexample leads to at least one new state in \(H\)
  - \(n - 1\) counterexamples are sufficient

- **Query complexity dominated by size of observation table (1 test case per cell)**
  - \(kn + 1\) rows
  - 1 suffix added per counterexample \(\Rightarrow\) at most \(n\) columns
  \(\Rightarrow\) size of table in \(O(kn^2)\)
Size of observation table: $O(kn^2)$

- Plus: counterexample analysis (Rivest & Schapire)
  - Binary search: $O(\log m)$ queries per counterexample
  - at most $n - 1$ counterexamples $\Rightarrow$ total $O(n \log m)$ queries for counterexample analysis

- Total query complexity: $O(kn^2 + n \log m)$

- Known lower bound [Balcázar et al., 1997]: $\Omega(kn^2)$
Discrimination Trees

Redundancies in Observation Table

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ba</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bb</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>aa</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ab</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Redundancies in Observation Table

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ba</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bb</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>aa</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ab</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Value 0 in ε-column **sufficient** to distinguish [b] from other two states
### Discrimination Trees

**Redundancies in Observation Table**

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1 1</td>
<td></td>
</tr>
<tr>
<td>ba</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>bb</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>aa</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ab</td>
<td>1 1</td>
<td></td>
</tr>
</tbody>
</table>

- Value 0 in ε-column **sufficient** to distinguish [b] from other two states

![Discrimination Tree Diagram]
### Discrimination Trees

#### Redundancies in Observation Table

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$b$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$ba$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$bb$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$aa$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$ab$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Value 0 in $\varepsilon$-column **sufficient** to distinguish $[b]$ from other two states.

![Discrimination Tree Diagram]

**Explanation:**

- The diagram represents a discrimination tree where $\varepsilon$, $b$, and $a$ are the symbols, and $[\varepsilon]$, $[a]$, and $[b]$ are the states.
- The transitions are labeled with symbols that cause a change in state.
- The value 0 in the $\varepsilon$ column is sufficient to distinguish $[b]$ from other states.
**Discrimination Trees**

**Redundancies in Observation Table**

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ba</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>bb</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>aa</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ab</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Value 0 in ε-column **sufficient** to distinguish \([b]\) from other two states
- Discrimination Tree redundancy-free: exactly one distinguishing suffix for each pair of states (LCA)
Determining the target state of the $b$-transition of state $[a]$: 

- Access sequence of transition: $ab$
- "Sift" $ab$ into discrimination tree

Diagram:

1. **State $[\varepsilon]$**
   - Transitions:
     - $a$ to $[a]$
     - $b$ to $[b]$

2. **State $[a]$**
   - Transitions:
     - $a$ to $[a]$
     - $b$ to $[b]$

3. **State $[b]$**

Discrimination Trees: Sifting
Determining the target state of the $b$-transition of state $[a]$:  

- Access sequence of transition: access seq. of state + symbol (in this case: $ab$)
Determining the target state of the $b$-transition of state $[a]$: 

- Access sequence of transition: access seq. of state + symbol (in this case: $ab$)
- “Sift” $ab$ into discrimination tree
Determining the target state of the $b$-transition of state $[a]$:  

- Access sequence of transition: access seq. of state + symbol  
  (in this case: $ab$)  
- “Sift” $ab$ into discrimination tree
Determining the target state of the $b$-transition of state $[a]$: 

- Access sequence of transition: access seq. of state + symbol (in this case: $ab$)
- “Sift” $ab$ into discrimination tree
Determining the target state of the $b$-transition of state $[a]$:

- Access sequence of transition: access seq. of state + symbol (in this case: $ab$)
- “Sift” $ab$ into discrimination tree

\[ \lambda(ab \cdot \varepsilon) = 1 \]
Determining the target state of the $b$-transition of state $[a]$:

- Access sequence of transition: access seq. of state + symbol (in this case: $ab$)
- “Sift” $ab$ into discrimination tree
Determining the target state of the $b$-transition of state $[a]$:

- Access sequence of transition: access seq. of state + symbol
  (in this case: $ab$)
- “Sift” $ab$ into discrimination tree
Determining the target state of the $b$-transition of state $[a]$: 

- Access sequence of transition: access seq. of state + symbol (in this case: $ab$)
- “Sift” $ab$ into discrimination tree
Determining the target state of the $b$-transition of state $[a]$:

- Access sequence of transition: access seq. of state + symbol (in this case: $ab$)
- “Sift” $ab$ into discrimination tree
Determining the target state of the $b$-transition of state $[a]$: 

- Access sequence of transition: access seq. of state + symbol (in this case: $ab$)
- “Sift” $ab$ into discrimination tree
Discrimination Trees: Refinement

Reminder: Counterexample decomposition yields \( u, u' \in U, a \in \Sigma, v \in \Sigma^* \)
- Observation Table: simply add \( v \) to columns
- Does not work for discrimination trees (locality)!
Remainder: Counterexample decomposition yields \(u, u' \in U, a \in \Sigma, v \in \Sigma^*\)

- Observation Table: simply add \(v\) to columns
- Does not work for discrimination trees (locality)!

Refinement: split leaf corresponding to \(u' = \varepsilon\)
Reminder: Counterexample decomposition yields $u, u' \in \mathcal{U}, a \in \Sigma, v \in \Sigma^*$

- Observation Table: simply add $v$ to columns
- Does not work for discrimination trees (locality)!

Refinement: split leaf corresponding to $u' = \varepsilon$
Reminder: Counterexample decomposition yields $u, u' \in \mathcal{U}$, $a \in \Sigma$, $v \in \Sigma^*$

- Observation Table: simply add $v$ to columns
- Does not work for discrimination trees (locality)!

Refinement: split leaf corresponding to $u' = \varepsilon$
- Suffix at new inner node: $v = b$
Discrimination Trees: Refinement

Reminder: Counterexample decomposition yields $u, u' \in U$, $a \in \Sigma$, $v \in \Sigma^*$

- Observation Table: simply add $v$ to columns
- Does not work for discrimination trees (locality)!

Refinement: split leaf corresponding to $u' = \varepsilon$

- Suffix at new inner node: $v = b$
- New children (leaves): $[u']$ and new state $[ua] (u = \varepsilon)$
Spanning Tree Hypothesis

Let $H$ be a graph with states $\varepsilon$, $a$, $b$, and transitions $a$, $b$, $a, b$.

Observation: new states derived from transitions
Spanning Tree Hypothesis

Observation: new states derived from transitions

- i.e., access sequence of the form $ua$, where $u \in \mathcal{U}$, $a \in \Sigma$

$\Rightarrow \mathcal{U}$ stays prefix-closed
Spanning Tree Hypothesis

Observation: new states derived from transitions
- i.e., access sequence of the form $ua$, where $u \in \mathcal{U}, a \in \Sigma$

$\Rightarrow \mathcal{U}$ stays prefix-closed

Observation 2: $\mathcal{U} \setminus \{\varepsilon\}$ are transition access sequences
Spanning Tree Hypothesis

\[ \mathcal{H} \]

[Diagram of a graph with states \([\varepsilon]\), \([a]\), \([b]\) and transitions labeled with \(a\), \(b\) and \(a, b\).

Observation: new states derived from transitions

- i.e., access sequence of the form \(ua\), where \(u \in \mathcal{U}\), \(a \in \Sigma\)

\[ \Rightarrow \mathcal{U} \text{ stays prefix-closed} \]

Observation 2: \(\mathcal{U} \setminus \{\varepsilon\}\) are transition access sequences

- \(\mathcal{U}\) prefix-closed \(\Rightarrow\) transitions in \(\mathcal{U} \setminus \{\varepsilon\}\) form spanning tree

\[ \Rightarrow \text{Spanning tree can be used to represent } \mathcal{U} \]
Practical Challenges

Main practical challenge: Realizing membership and equivalence queries

- Need to fulfill certain criteria: independence, determinism, . . .

But first: DFAs not adequate for modeling reactive systems

- Better-suited model: Mealy machines
Mealy machines

\[ A \]

- \( q_0 \) to \( q_1 \): \( b/x \)
- \( q_0 \) to \( q_2 \): \( a/y \)
- \( q_1 \) to \( q_2 \): \( a/y \)
- \( q_2 \) to \( q_1 \): \( a/x, b/y \)

Output function:
\[ \lambda : \Sigma^* \rightarrow \Omega^* \]

\( \Omega \) is the output alphabet, \( \Omega = \{ x, y \} \)

DFA learning can be adapted to Mealy machines in a straightforward way.
Mealy machines

- No accepting/rejecting states, but **transition outputs**
Mealy machines

- No accepting/rejecting states, but transition outputs
- Output function: $\lambda : \Sigma^* \rightarrow \Omega^*$ ($\Omega$: output alphabet, $\Omega = \{x, y\}$)
Mealy machines

- No accepting/rejecting states, but transition outputs
- Output function: $\lambda : \Sigma^* \rightarrow \Omega^*$ ($\Omega$: output alphabet, $\Omega = \{x, y\}$)
- DFA learning can be adapted to Mealy machines in a straightforward way
Realizing Membership Queries

- Membership Queries \( \hat{=} \) test cases
- Step 1: define learning alphabet \( \Sigma \)
  - e.g., API calls, communication primitives etc.
- Step 2: define *semantics* for each symbol
Realizing Membership Queries

- Membership Queries $\hat{=} \text{test cases}$
- Step 1: define learning alphabet $\Sigma$
  - e.g., API calls, communication primitives etc.
- Step 2: define semantics for each symbol

Challenges:
- Queries need to be independent
  - system needs to be reset (expensive, sometimes not feasible)
  - “fresh” session, user, ... might be sufficient
Realizing Membership Queries

- Membership Queries \( \hat{=} \) test cases
- Step 1: define learning alphabet \( \Sigma \)
  - e.g., API calls, communication primitives etc.
- Step 2: define semantics for each symbol

Challenges:
- Queries need to be independent
  - system needs to be reset (expensive, sometimes not feasible)
  - “fresh” session, user, ... might be sufficient
- Response needs to be deterministic
  - suitable output abstraction (e.g., no absolute timestamps etc.)
Realizing Membership Queries

- Membership Queries $\hat{=} \text{test cases}$
- Step 1: define learning alphabet $\Sigma$
  - e.g., API calls, communication primitives etc.
- Step 2: define semantics for each symbol

Challenges:
- Queries need to be independent
  - system needs to be reset (expensive, sometimes not feasible)
  - “fresh” session, user, … might be sufficient
- Response needs to be deterministic
  - suitable output abstraction (e.g., no absolute timestamps etc.)
- Data-flow aspects need to be addressed
  - e.g., actions need session id, which is returned upon login etc.
Mapper

- Abstracts from lower-level system details
- Enables finite-state, finite-alphabet models
Realizing Equivalence Queries

Earlier: counterexamples trigger refinement, but how to obtain counterexamples?

- Abstract formulation: equivalence queries
Realizing Equivalence Queries

Earlier: counterexamples trigger refinement, but how to obtain counterexamples?

- Abstract formulation: equivalence queries
- Problem: realizing equivalence queries requires detailed knowledge about the system’s behavior
  - ... but obtaining this is the whole point in using learning ...
Realizing Equivalence Queries

Earlier: counterexamples trigger refinement, but how to obtain counterexamples?

- Abstract formulation: **equivalence queries**
- **Problem:** realizing equivalence queries requires detailed knowledge about the system’s behavior
  - ...but obtaining this is the whole point in using learning ...
- In practice: only **approximated** equivalence queries are possible
  - i.e., approximate equivalence queries through membership queries
Approximating Equivalence Queries

- Classical approach: conformance testing methods
  - W-method, Wp-method, UIO method etc.
  - relative guarantees (e.g., assuming the target system has at most $\Delta n$ additional states) at the cost of an exponential number of membership queries
Approximating Equivalence Queries

- Classical approach: conformance testing methods
  - W-method, Wp-method, UIO method etc.
  - relative guarantees (e.g., assuming the target system has at most $\Delta n$ additional states) at the cost of an exponential number of membership queries

- Often more viable: log-file analysis or monitoring
  $\Rightarrow$ “life-long learning”
Combining Learning and Monitoring

Target System

Clients

Δ

Model Validator

Membership Queries

Counterexample

Learner

Monitoring Data (traces)

Hypothesis

Executions may run for a long time ⇒ long counterexamples
Combining Learning and Monitoring

- Executions may run for a long time ⇒ long counterexamples
Counterexamples through Monitoring:

```
chmod_ro
open
read
close

chmod_rw
open
read, write
close
```

Introduction  Active Automata Learning  Automata Learning in Practice  The TTT Algorithm  LearnLib
Counterexamples through Monitoring:

```
prefix

chmod_ro

open

close

read

chmod_rw

open

close

read, write

chmod_ro

close

read

suffix

chmod_rw

open

close

read
```

"Optimal" counterexample chmod rw open write
Counterexamples through Monitoring:

prefix
open read close open read close chmod_rw

suffix
open read close open read close open write

\textit{Example}

- \texttt{open}
- \texttt{close}
- \texttt{read}
- \texttt{chmod}

\texttt{chmod}\_\texttt{ro}
\texttt{chmod}\_\texttt{rw}

- \texttt{chmod}\_\texttt{ro}
- \texttt{chmod}\_\texttt{rw}
- \texttt{chmod}\_\texttt{ro}
- \texttt{chmod}\_\texttt{rw}

- \texttt{chmod}\_\texttt{ro}
- \texttt{chmod}\_\texttt{ro}
- \texttt{chmod}\_\texttt{ro}

- \texttt{chmod}\_\texttt{ro}
- \texttt{chmod}\_\texttt{ro}
- \texttt{chmod}\_\texttt{ro}
- \texttt{chmod}\_\texttt{ro}

- \texttt{open}
- \texttt{read}
- \texttt{close}
- \texttt{open}
- \texttt{read}
- \texttt{close}
- \texttt{chmod}\_\texttt{rw}
- \texttt{open}
- \texttt{read}
- \texttt{close}
- \texttt{open}
- \texttt{read}
- \texttt{close}
- \texttt{chmod}\_\texttt{rw}
- \texttt{open}
- \texttt{read}
- \texttt{close}
- \texttt{open}
- \texttt{read}
- \texttt{close}
- \texttt{open}
- \texttt{write}

- \texttt{open}
- \texttt{read}
- \texttt{close}
- \texttt{open}
- \texttt{read}
- \texttt{close}
- \texttt{chmod}\_\texttt{rw}
- \texttt{open}
- \texttt{read}
- \texttt{close}
- \texttt{open}
- \texttt{read}
- \texttt{close}
- \texttt{chmod}\_\texttt{rw}
- \texttt{open}
- \texttt{read}
- \texttt{close}
- \texttt{open}
- \texttt{read}
- \texttt{close}
- \texttt{open}
- \texttt{write}
Counterexamples through Monitoring:

```
prefix
open read close open read close chmod_rw
```

```
open read close open read close chmod rw
```

```
suffix
open read close open read close open write
```

“Optimal” counterexample chmod_rw open write
The Problem of Long Counterexamples

- Counterexample decomposition: suffix $v$ extracted from counterexample
- Binary search: no guarantees wrt. length of this suffix (might not even be the shortest such suffix)

$\Rightarrow$ length of suffix $v$ only bounded by $m$ (length of counterexample)

So what is the problem with long suffixes?
The Problem of Long Suffixes

Consequence: long membership queries

Typically: time required for performing membership query linear in its length

Even worse: impact not “local” but persists throughout entire process
The Problem of Long Suffixes

Consequence: long membership queries

Typically: time required for performing membership query linear in its length

Even worse: impact not "local" but persists throughout entire process
The Problem of Long Suffixes

- Consequence: long membership queries
  - $aaaabbaabba, bbaabbaabba, \ldots$
  - Typically: time required for performing membership query linear in its length
- Even worse: impact not “local” but persists throughout entire process
Idea: “normalize” suffixes to establish suffix-closedness
- Length of suffixes then bounded by $n$, organized in Trie
- Spanning Tree, Discrimination Tree, Suffix Trie
- Linear (optimal) space complexity
TTT – Another Example
Main idea: mark new suffixes (i.e., extracted from a counterexample) as temporary

- Final suffixes only above any temporary suffixes
- Maximal subtree w/o final suffixes: block

Finalize temporary suffixes by replacing them with newly constructed ones (preserving suffix closedness)

- Replace block roots only
Find:

- 2 states $q, q'$ in the same block
Find:

- 2 states $q, q'$ in the same block
Find:

- 2 states $q, q'$ in the same block
- symbol $a \in \Sigma$ s.t. $a$-successors of $q$ and $q'$ are in different blocks
Discriminator Finalization – Idea

Find:

- 2 states $q, q'$ in the same block
- symbol $a \in \Sigma$ s.t. $a$-successors of $q$ and $q'$ are in different blocks
Find:

- 2 states \( q, q' \) in the same block
- symbol \( a \in \Sigma \) s.t. \( a \)-successors of \( q \) and \( q' \) are in different blocks
Discriminator Finalization – Idea

Find:
- 2 states \( q, q' \) in the same block
- symbol \( a \in \Sigma \) s.t. \( a \)-successors of \( q \) and \( q' \) are in different blocks
  - then: LCA of successors is labeled with final suffix \( v \)
Find:

- 2 states $q, q'$ in the same block
- symbol $a \in \Sigma$ s.t. $a$-successors of $q$ and $q'$ are in different blocks
  - then: LCA of successors is labeled with final suffix $v$
Find:

- 2 states $q, q'$ in the same block
- symbol $a \in \Sigma$ s.t. $a$-successors of $q$ and $q'$ are in different blocks
  
  - then: LCA of successors is labeled with final suffix $v$
  - $av$ separates $q$ and $q' \Rightarrow$ new final suffix
Find:

- 2 states $q, q'$ in the same block
- symbol $a \in \Sigma$ s.t. $a$-successors of $q$ and $q'$ are in different blocks
  - then: LCA of successors is labeled with final suffix $v$
  - $av$ separates $q$ and $q' \Rightarrow$ new final suffix
Q: Can we *always* find such $q, q', a$?
Q: Can we always find such $q, q', a$?
A: Not always - but then our hypothesis is wrong!
Hypothesis **contradicts** correct information $\lambda(a \cdot aabbb) = 1$ (stored in DT)

- **Non-canonical hypothesis**
- Analyze counterexample $a \cdot aabbb/1 \ldots$
Hypothesis **contradicts** correct information $\lambda(a \cdot aabbb) = 1$  
(Stored in DT)

- Non-canonical hypothesis
- Analyze counterexample $a \cdot aabbb/1 \ldots$
TTT – Some Results (1)

![Graphs showing query and symbol counts for sched4 with different algorithms: RS, KV, DT, and TTT.](image)

**Introduction**

Active Automata Learning

**Automata Learning in Practice**

The **TTT Algorithm**

LearnLib
TTT – Some Results (2)
### TTT – Some Results (3)

| Alg. | Random-10-100 \((\hat{n}/2 \leq |w| \leq 2\hat{n})\) | Random-10-100 \(|w| = 500\) |
|------|------------------------------------------------|--------------------------------|
| \(L^*\) | 89518 (±31%) 8515147 (±58%) 3 (±0) | 104428 (±43%) 25831586 (±44%) 1 (±0) |
| SUFFIX1BY1 | 29529 (±12%) 185567 (±19%) 4 (±1) | 28817 (±11%) 201714 (±19%) 7 (±1) |
| RS | 28051 (±7%) 564633 (±59%) 3 (±0) | 30286 (±13%) 1691647 (±81%) 1 (±0) |
| KV | 22149 (±6%) 1017847 (±23%) 12 (±2) | 24128 (±8%) 2370053 (±25%) 27 (±6) |
| DT | 15203 (±0%) 195958 (±14%) 7 (±1) | 16624 (±5%) 1154253 (±96%) 1 (±1) |
| TTT | 14930 (±0%) 123742 (±8%) 8 (±2) | 15142 (±0%) 157407 (±23%) 2 (±1) |

| Alg. | Random-20-200 \((\hat{n}/2 \leq |w| \leq 2\hat{n})\) | Random-20-200 \(|w| = 500\) |
|------|------------------------------------------------|--------------------------------|
| \(L^*\) | 181389 (±28%) 16854213 (±53%) 3 (±0) | 211490 (±38%) 52334184 (±40%) 1 (±0) |
| SUFFIX1BY1 | 59655 (±11%) 333687 (±18%) 4 (±1) | 60977 (±12%) 377621 (±20%) 7 (±1) |
| RS | 56158 (±7%) 906177 (±57%) 2 (±0) | 59250 (±10%) 1733102 (±70%) 1 (±0) |
| KV | 36794 (±3%) 1270735 (±20%) 11 (±2) | 37757 (±4%) 2654647 (±25%) 25 (±6) |
| DT | 31030 (±0%) 375067 (±13%) 6 (±1) | 32987 (±4%) 1637273 (±73%) 1 (±0) |
| TTT | 30820 (±0%) 230955 (±9%) 7 (±1) | 31086 (±0%) 278940 (±15%) 1 (±1) |

| Alg. | SCHED4 \((\hat{n}/2 \leq |w| \leq 2\hat{n})\) | SCHED4 \(|w| = 500\) |
|------|--------------------------------|--------------------------------|
| \(L^*\) | 68377 (±16%) 3879529 (±27%) 12 (±0) | 140906 (±35%) 35758545 (±35%) 2 (±0) |
| SUFFIX1BY1 | 46658 (±33%) 2188119 (±39%) 28 (±4) | —† — | —† — |
| RS | 11289 (±17%) 388726 (±25%) 36 (±6) | —† — | —† — |
| KV | 15921 (±9%) 735843 (±14%) 71 (±3) | 40181 (±15%) 7548726 (±18%) 18 (±2) |
| DT | 4321 (±8%) 165040 (±20%) 74 (±5) | 10647 (±1%) 3758972 (±5%) 18 (±2) |
| TTT | 3429 (±11%) 124984 (±37%) 73 (±3) | 2449 (±0%) 77177 (±6%) 18 (±2) |
Outline

1. Introduction
2. Active Automata Learning
3. Automata Learning in Practice
4. The TTT Algorithm
5. LearnLib
LearnLib

- Website: http://learnlib.de
- GitHub page: https://github.com/LearnLib/learnlib
  - Check out the Wiki for a quick setup guide
LearnLib

- Website: http://learnlib.de
- GitHub page: https://github.com/LearnLib/learnlib
  - Check out the Wiki for a quick setup guide

Demo after coffee break