LearnLib Tutorial

Malte Isberner[1], Bernhard Steffen[1], Falk Howar[2]

[1]: TU Dortmund University, Chair for Programming Systems, Germany
[2]: IPSSE / TU Clausthal, Germany

technische universität dortmund

fakultät für informatik
lehrstuhl für programmiersysteme

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

Outline

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

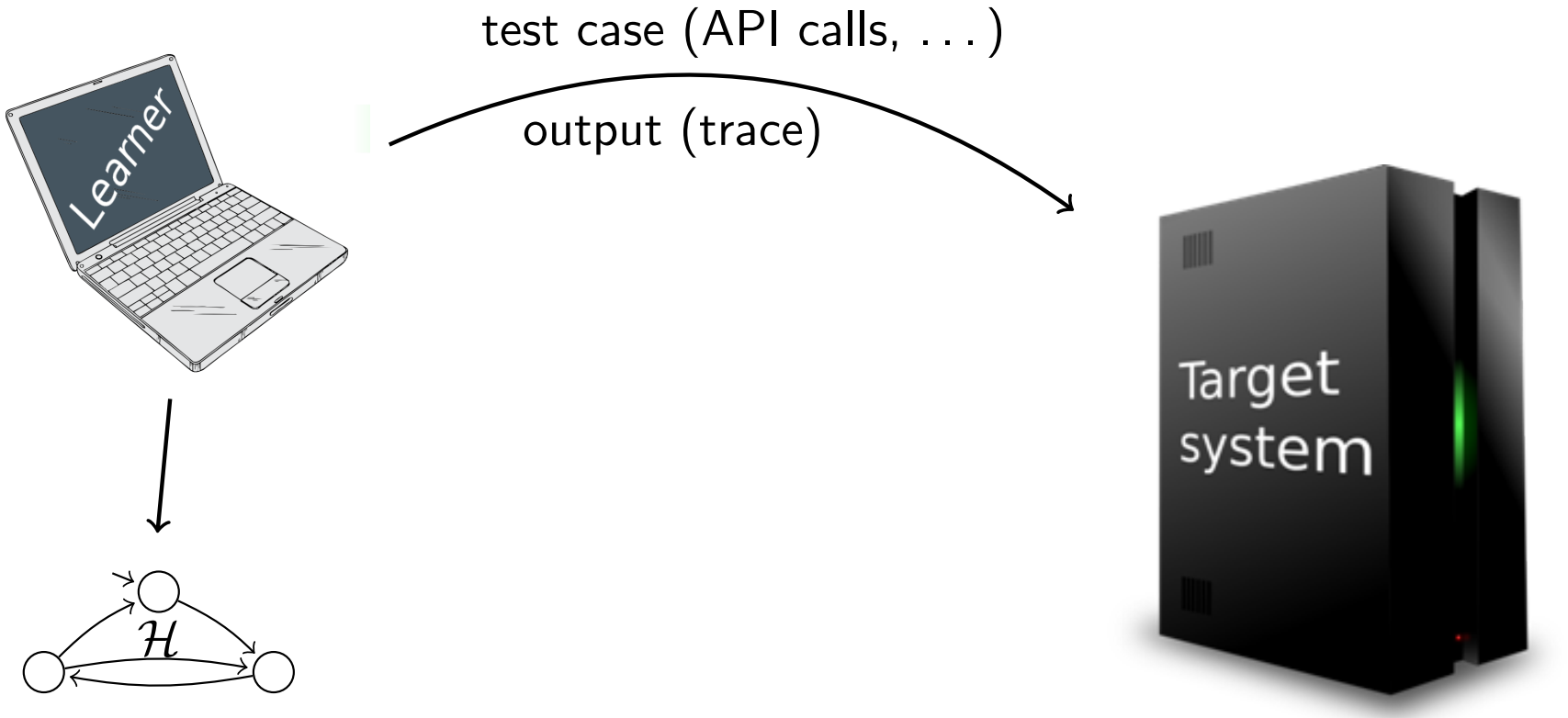
High-Level Overview

High-Level Overview

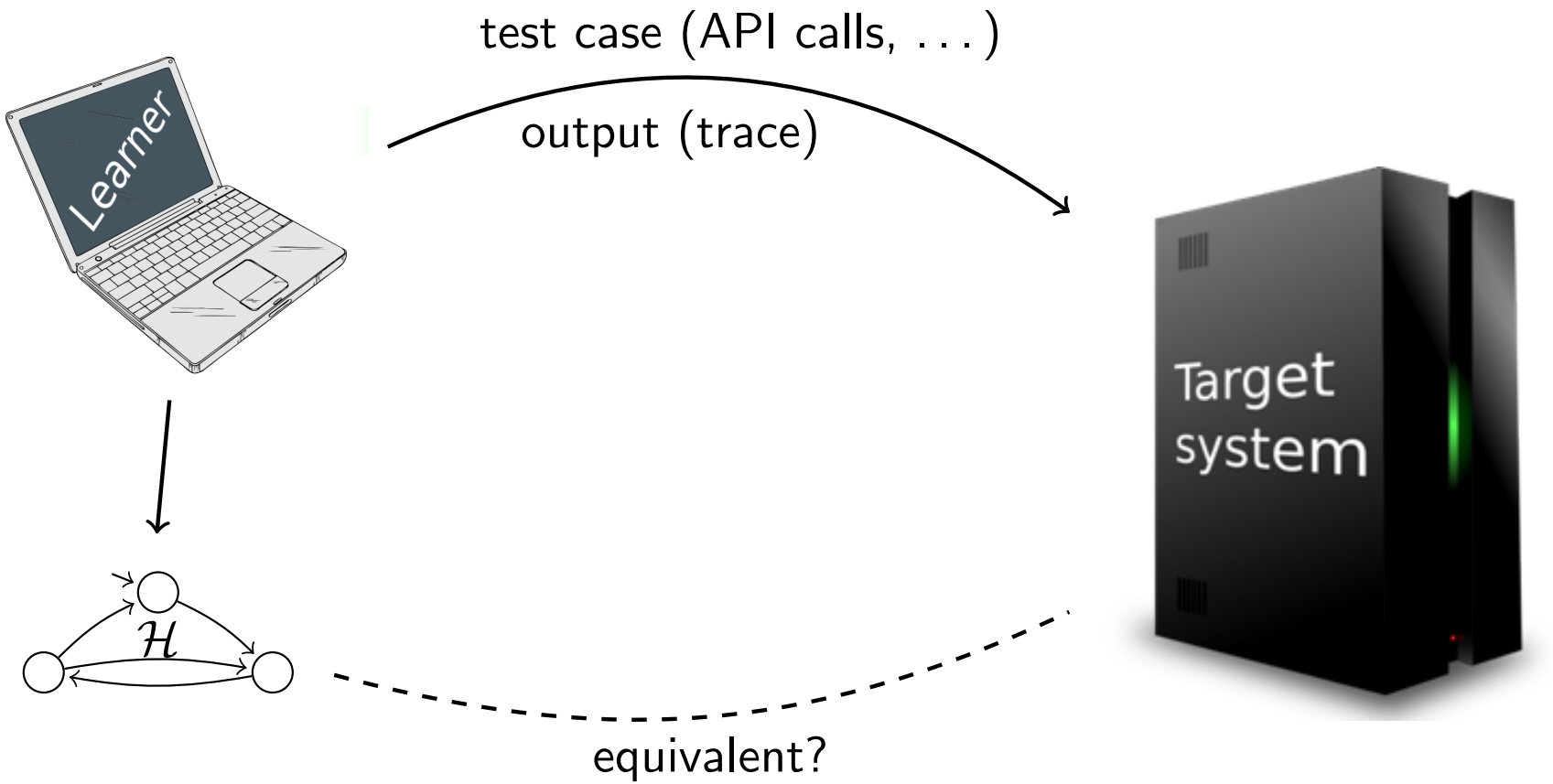
test case (API calls, . . .)

output (trace)

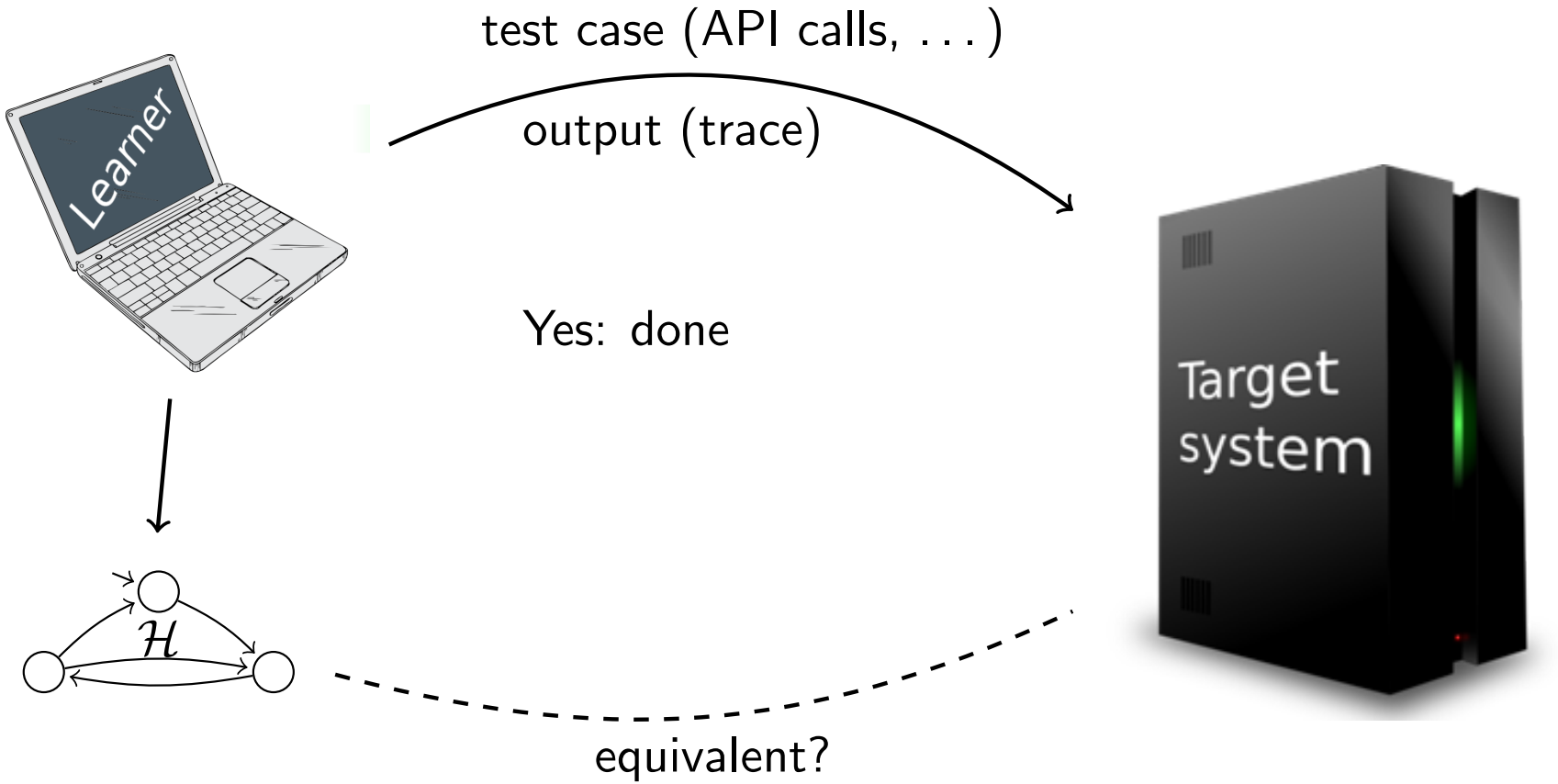
High-Level Overview



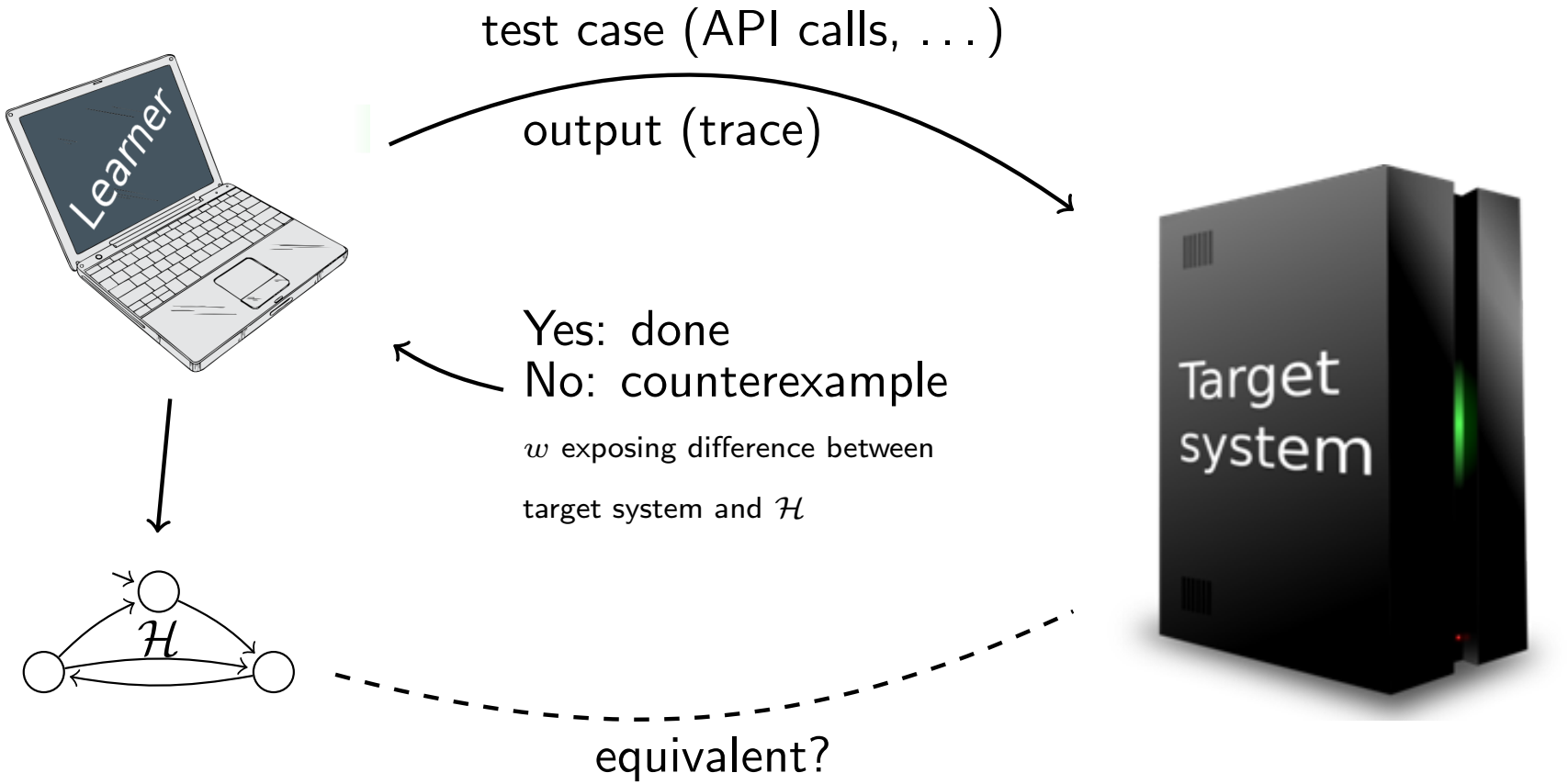
High-Level Overview



High-Level Overview



High-Level Overview



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!

Active Automata Learning

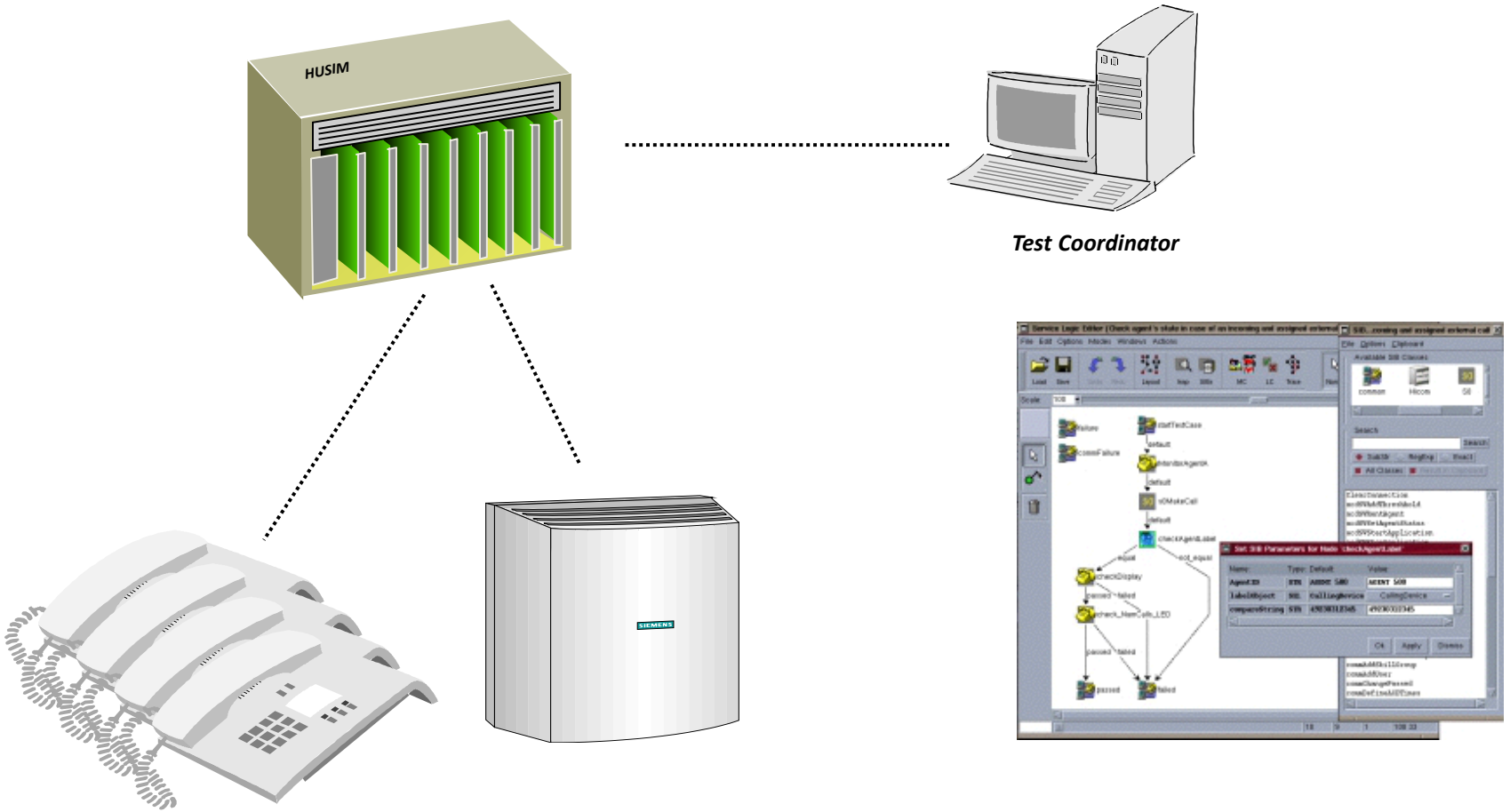
- Active Automata Learning: Construct (behavioral) models of (black-box) systems via testing
  - Typically: finite-state models (DFAs or Mealy machines)
  - "Inverse" of model-based testing: test-based modeling!
- Interaction with target systems via queries
  - Membership Queries (MQs): what is the response 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

- 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)?

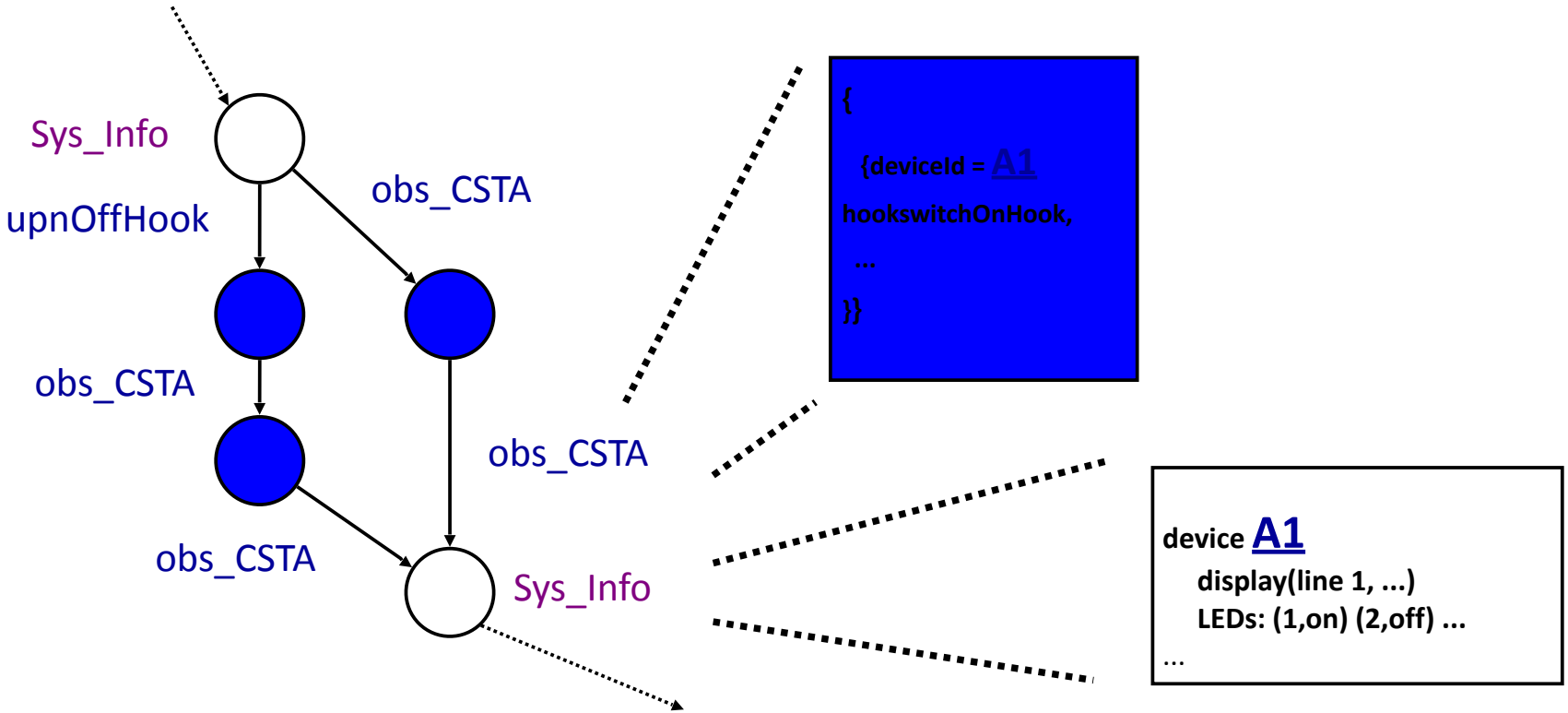
So what is this good for?

Testing Telephony Systems



- Hungar, Magaria, Steffen: Test-Based Model Generation for Legacy Systems (ITC 2003)

Testing Telephony Systems (2)

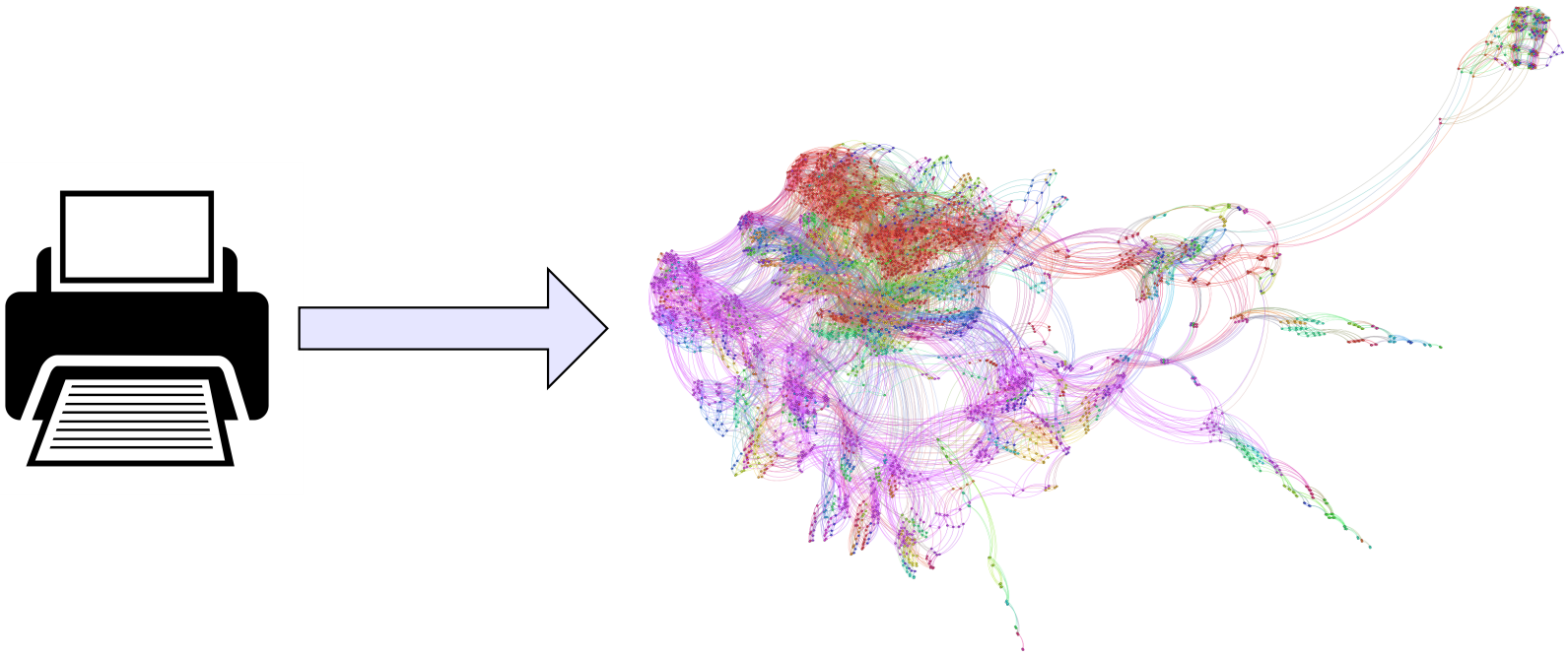


(small) learned models imposed
major test suite optimizations

Another Application: Learning Models of

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

Restrictions: System Requirements

What are the restrictions for applying active automata learning?

- Finite input alphabet Σ
- 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
  - Adaption to Mealy machines pretty straightforward

Technical Realization: Setup

Assumption: the behavior of the target system is modeled by a DFA $\mathcal{A}$

- Membership Queries ask if a word $w \in \Sigma^*$ is accepted by $\mathcal{A}$
- Notation: $\lambda(w) = 1$ iff w is accepted by $\mathcal{A}$ (otherwise 0) (λ: "output function")

$\Rightarrow$ Membership query $\hat{=}$ λ-evaluation

Technical Realization: Main Idea

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: $\mathcal{A}[u] \mathrel{\hat{=}}$ 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: $\mathcal{A}[u] \mathrel{\hat{=}}$ 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"

Technical Realization: Main Idea

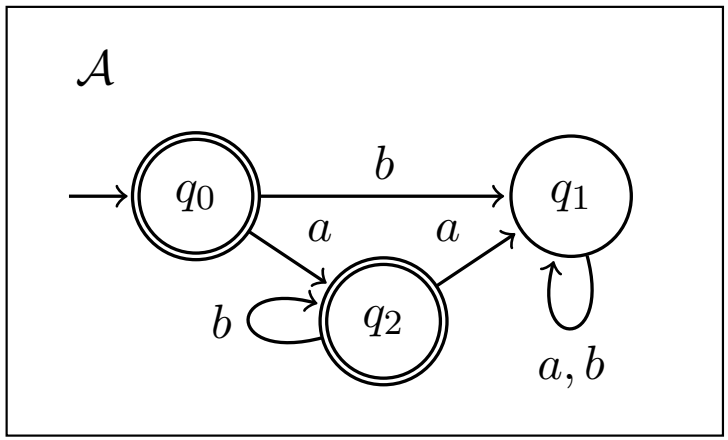
Main challenge: reasoning about structure (states and transitions) of unknown DFA $\mathcal{A}$

- Represent states of $\mathcal{A}$ by words $u \in \Sigma^*$ reaching them
  - Notation: $\mathcal{A}[u] \mathrel{\hat{=}}$ 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 Σ^*)

Realization: Observation Table

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

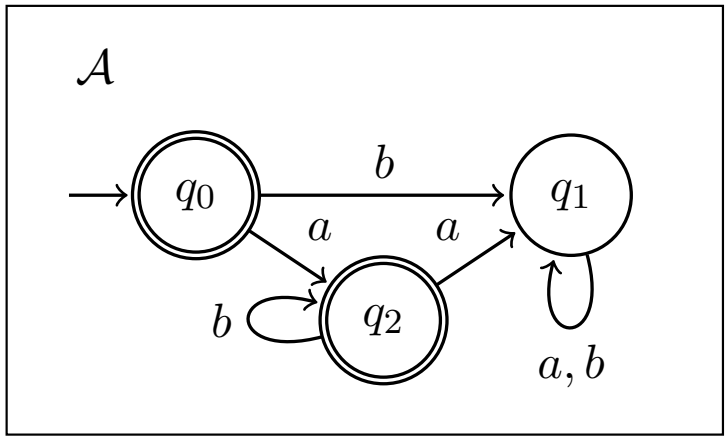
	ε
ε	1
b	0
a	1
ba	0
bb	0



Realization: Observation Table

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

	ε
ε	1
b	0
a	1
ba	0
bb	0

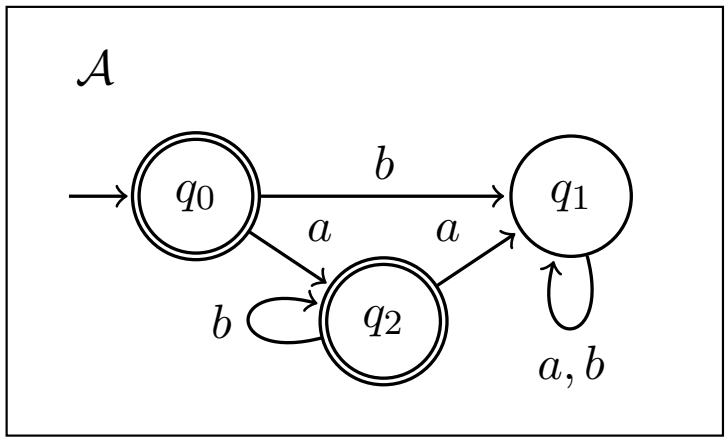


- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**

Realization: Observation Table

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

	ε
ε	1
b	0
a	1
ba	0
bb	0

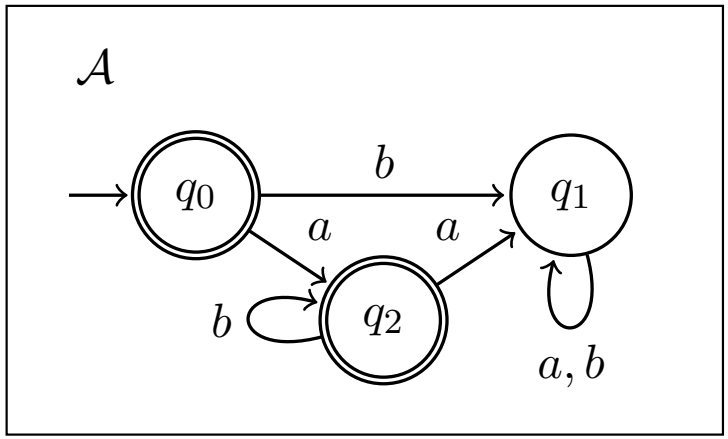


- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - Lower part: Long prefixes identify **transitions** (one-letter extensions)

Realization: Observation Table

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

	ε
ε	1
b	0
a	1
ba	0
bb	0

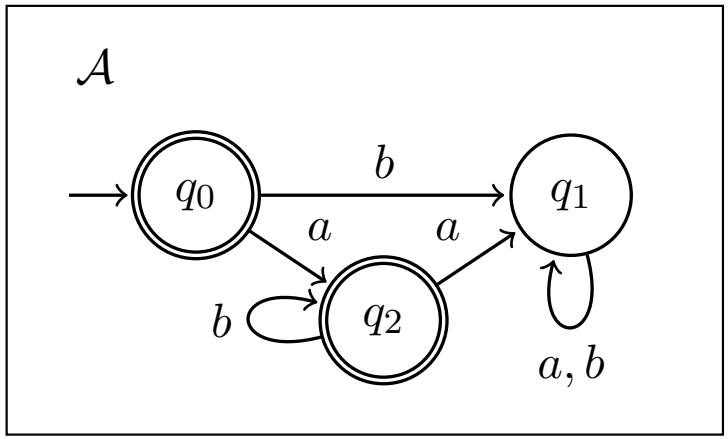


- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - Lower part: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes

Realization: Observation Table

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

	ε
ε	1
b	0
a	1
ba	0
bb	0



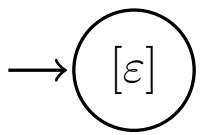
- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - Lower part: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
- Cell contents (row u, col. v): $\lambda(u \cdot v)$

Realization: Observation Table

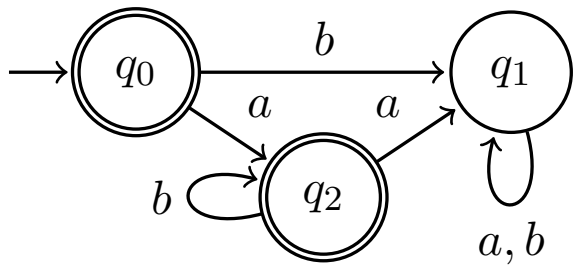
Data structure of L* (and similar algorithms): Observation Table

	ε
ε	1
b	0
a	1
ba	0
bb	0

$\mathcal{H}$



$\mathcal{A}$

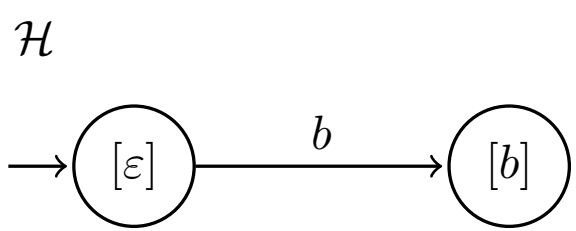


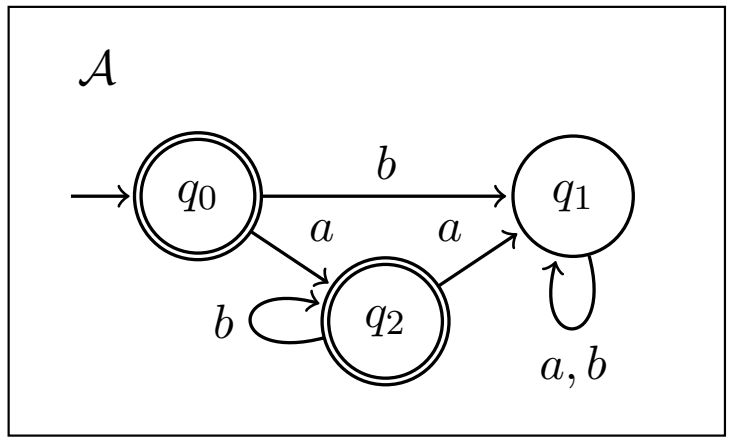
- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - Lower part: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
- Cell contents (row u, col. v): $\lambda(u \cdot v)$

Realization: Observation Table

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

	ε
ε	1
b	0
a	1
ba	0
bb	0



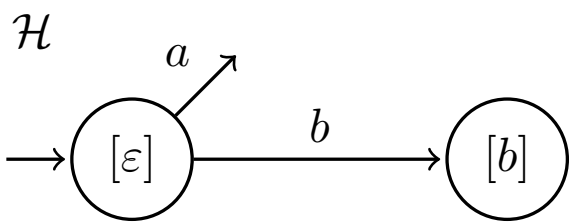


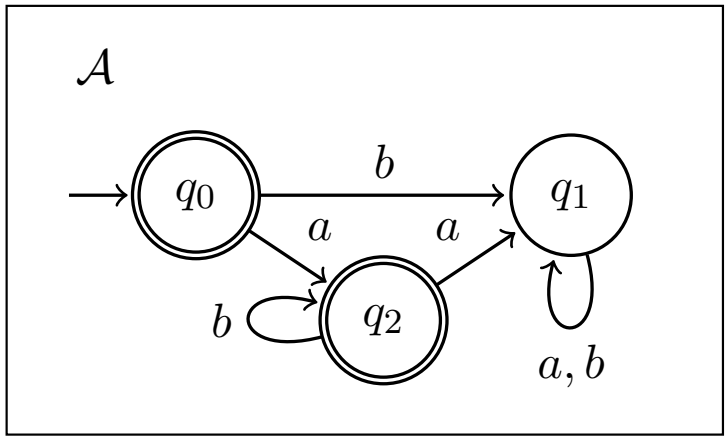
- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - Lower part: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
- Cell contents (row u, col. v): $\lambda(u \cdot v)$

Realization: Observation Table

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

	ε
ε	1
b	0
a	1
ba	0
bb	0



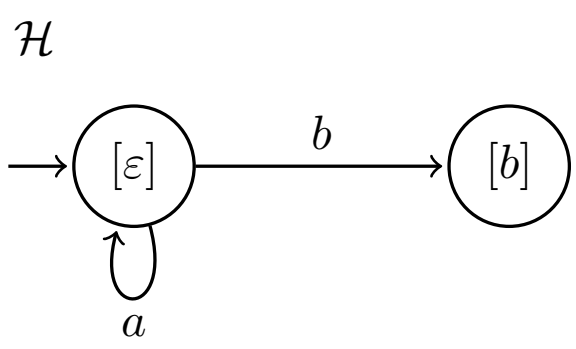


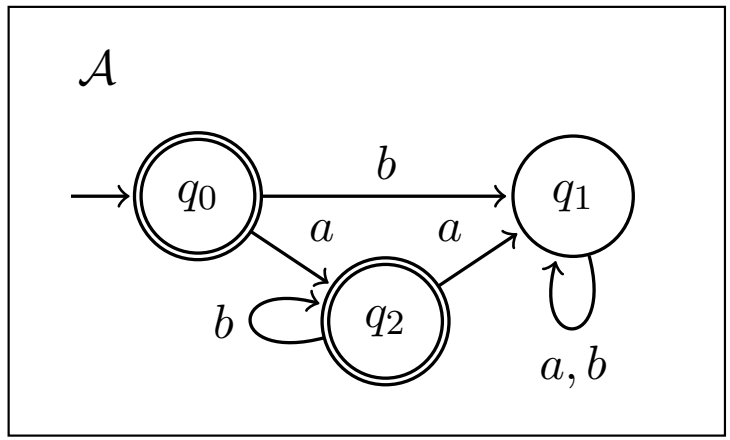
- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - Lower part: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
- Cell contents (row u, col. v): $\lambda(u \cdot v)$

Realization: Observation Table

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

	ε
ε	1
b	0
a	1
ba	0
bb	0



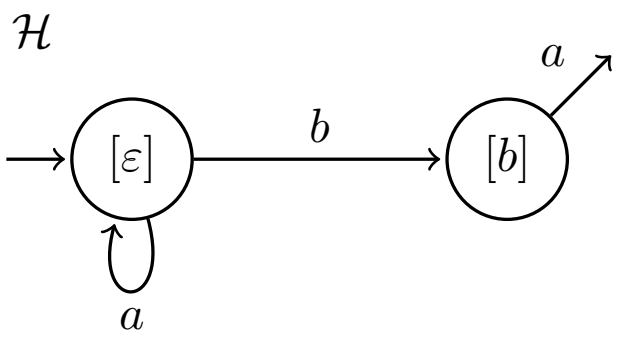


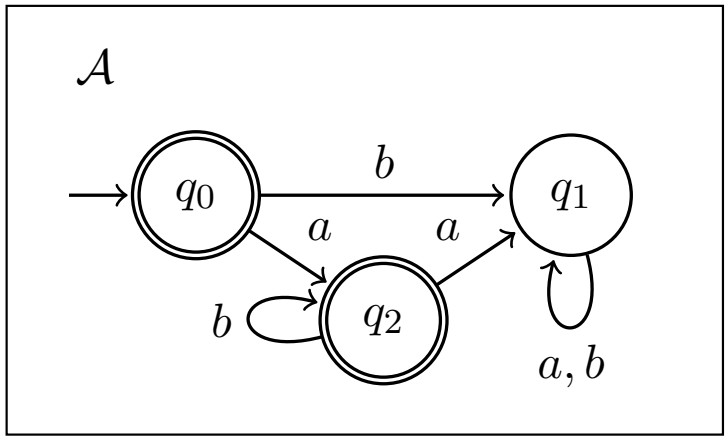
- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - Lower part: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
- Cell contents (row u, col. v): $\lambda(u \cdot v)$

Realization: Observation Table

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

	ε
ε	1
b	0
a	1
ba	0
bb	0



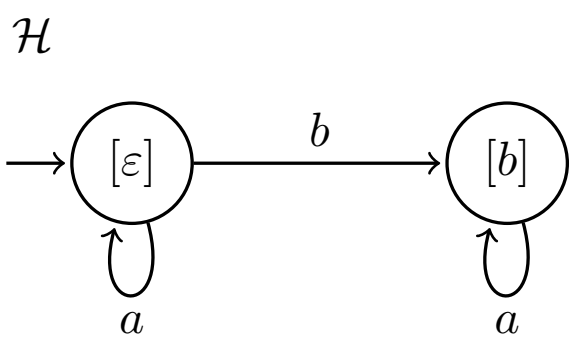


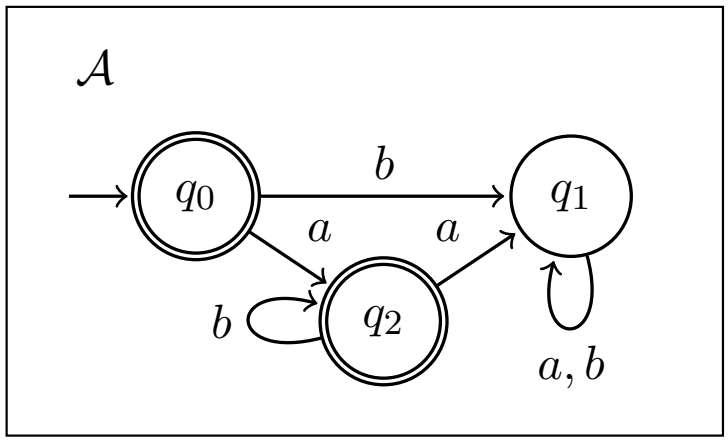
- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - Lower part: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
- Cell contents (row u, col. v): $\lambda(u \cdot v)$

Realization: Observation Table

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

	ε
ε	1
b	0
a	1
ba	0
bb	0



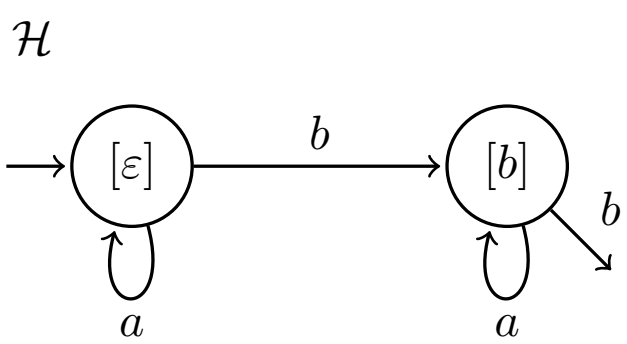


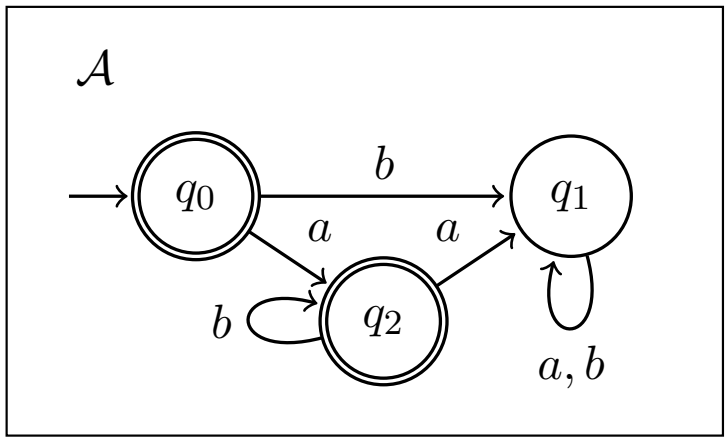
- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - Lower part: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
- Cell contents (row u, col. v): $\lambda(u \cdot v)$

Realization: Observation Table

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

	ε
ε	1
b	0
a	1
ba	0
bb	0



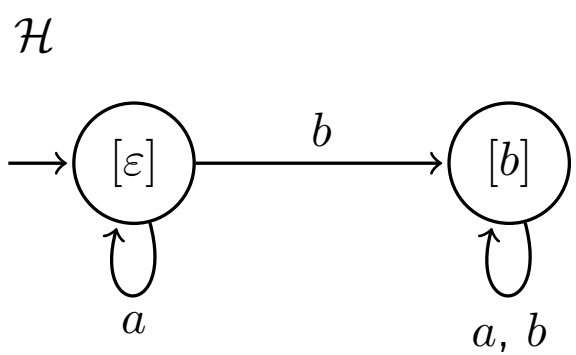


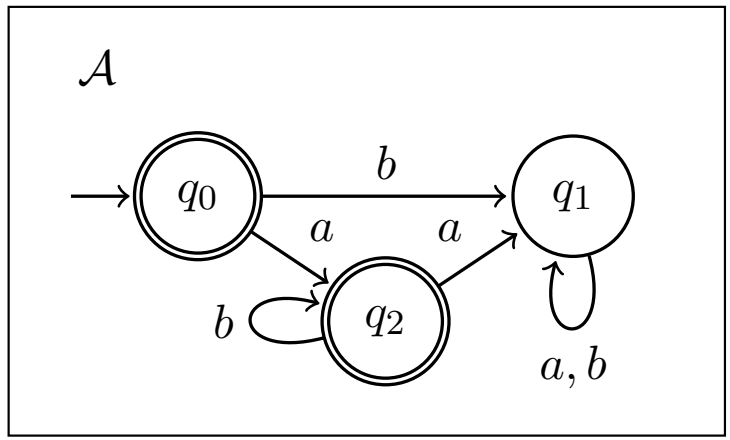
- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - Lower part: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
- Cell contents (row u, col. v): $\lambda(u \cdot v)$

Realization: Observation Table

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

	ε
ε	1
b	0
a	1
ba	0
bb	0



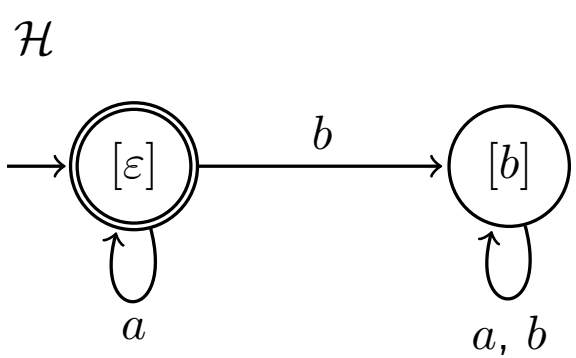


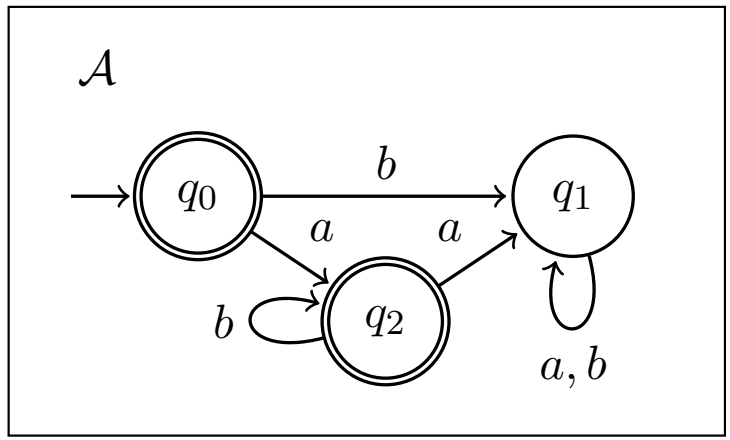
- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - Lower part: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
- Cell contents (row u, col. v): $\lambda(u \cdot v)$

Realization: Observation Table

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

	ε
ε	1
b	0
a	1
ba	0
bb	0

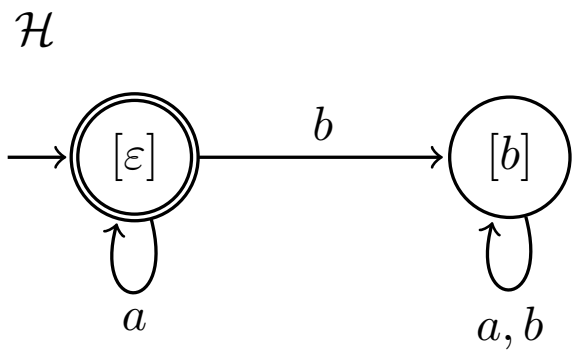


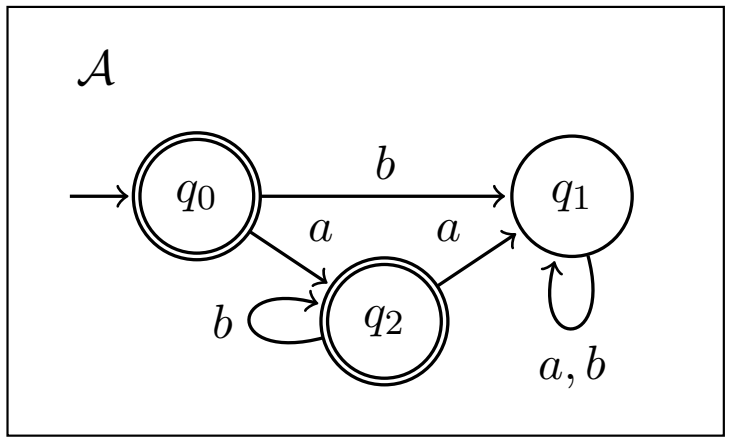


- Rows:
  - Upper part: Short prefixes ($\mathcal{U} = \{\varepsilon, b\}$) identify **states**
  - Lower part: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
- Cell contents (row u, col. v): $\lambda(u \cdot v)$

Refinement

	ε
ε	1
b	0
a	1
ba	0
bb	0

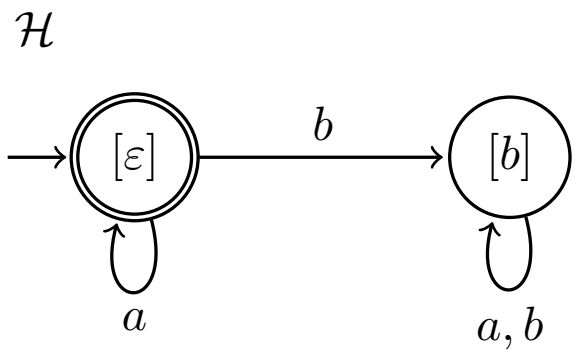


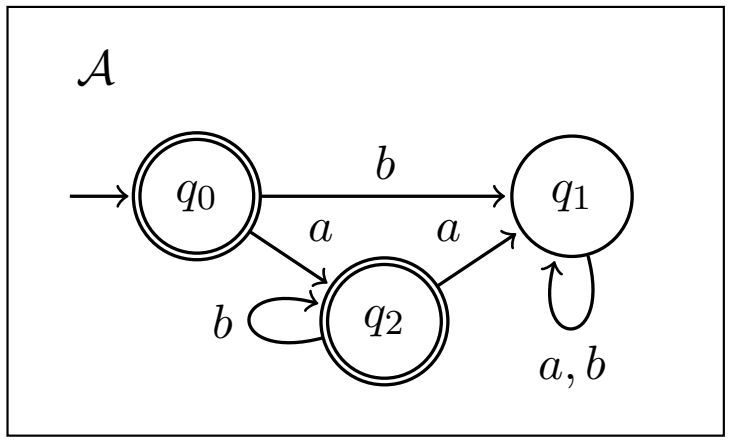


- $\mathcal{H}$ and $\mathcal{A}$ not equivalent!

Refinement

	ε
ε	1
b	0
a	1
ba	0
bb	0

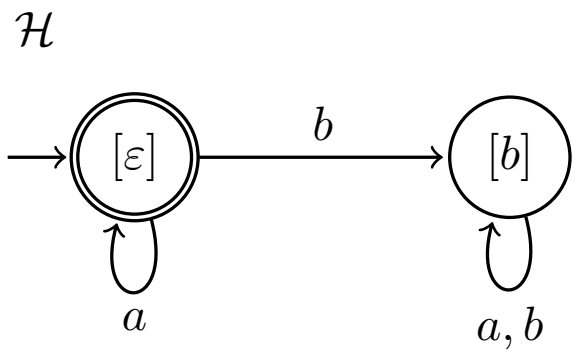


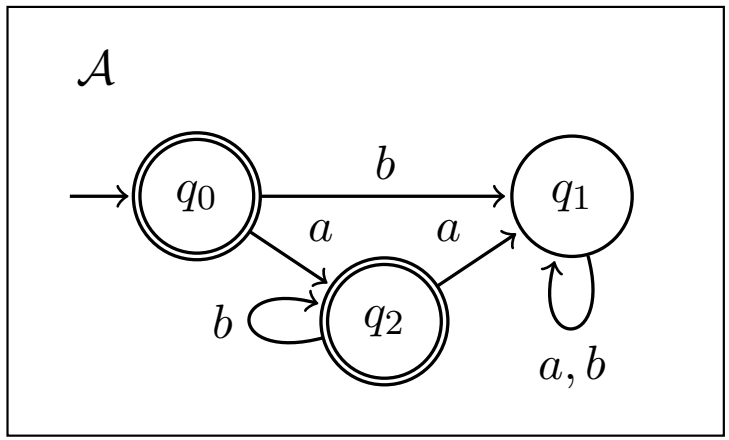


- $\mathcal{H}$ and $\mathcal{A}$ not equivalent!
- Reason: q_0 and q_2 in $\mathcal{A}$ have same future behavior wrt. ε

Refinement

	ε
ε	1
b	0
a	1
ba	0
bb	0

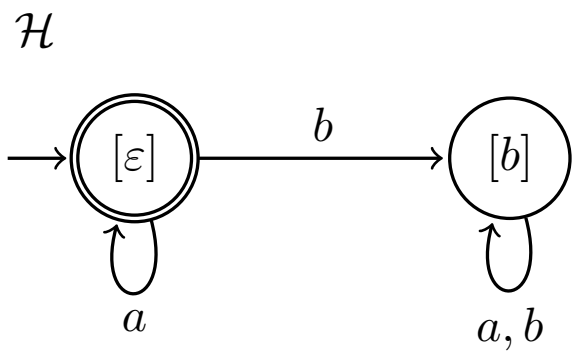


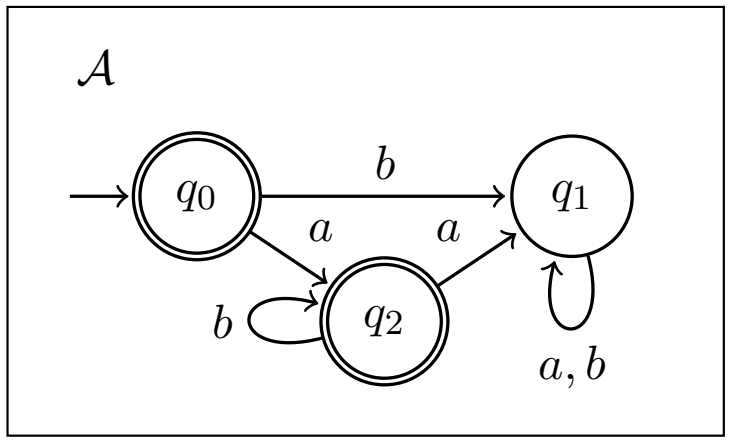


- $\mathcal{H}$ and $\mathcal{A}$ not equivalent!
- Reason: q_0 and q_2 in $\mathcal{A}$ have same future behavior wrt. ε
  - $\Rightarrow$ ε alone not sufficient for distinguishing q_0 (ε) and q_2 (a)
  - $\Rightarrow$ additional columns needed

Refinement

	ε	b
ε	1	
b	0	
a	1	
ba	0	
bb	0	

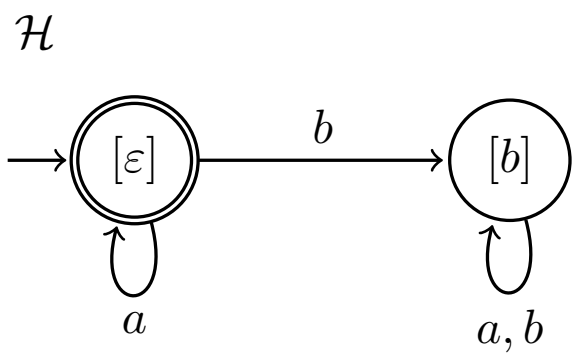


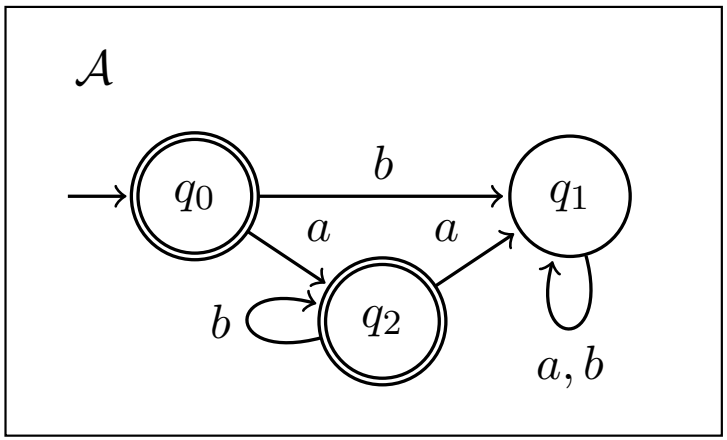


- $\mathcal{H}$ and $\mathcal{A}$ not equivalent!
- Reason: q_0 and q_2 in $\mathcal{A}$ have same future behavior wrt. ε
  - $\Rightarrow$ ε alone not sufficient for distinguishing q_0 (ε) and q_2 (a)
  - $\Rightarrow$ additional columns needed

Refinement

	ε	b
ε	1	0
b	0	0
a	1	1
ba	0	0
bb	0	0

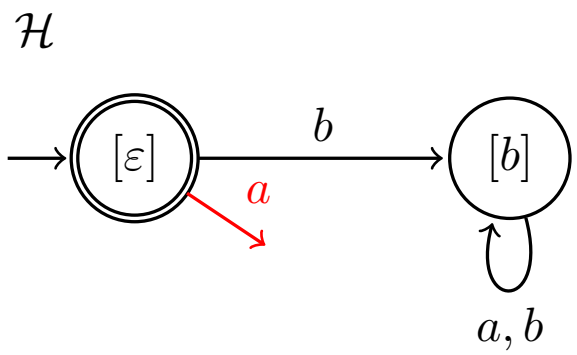


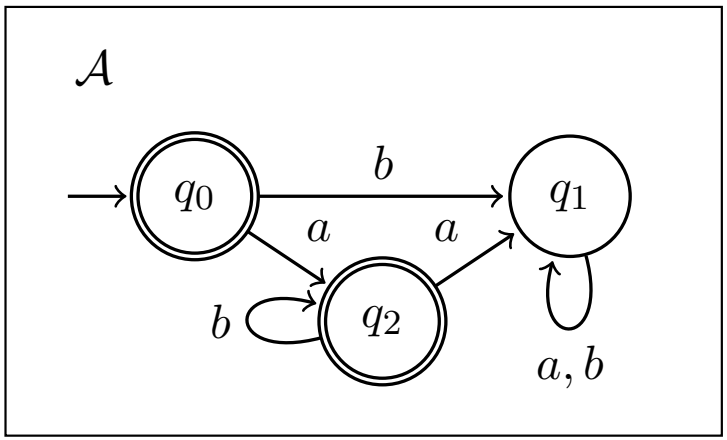


- $\mathcal{H}$ and $\mathcal{A}$ not equivalent!
- Reason: q_0 and q_2 in $\mathcal{A}$ have same future behavior wrt. ε
  - $\Rightarrow$ ε alone not sufficient for distinguishing q_0 (ε) and q_2 (a)
  - $\Rightarrow$ additional columns needed

Refinement

	ε	b
ε	1	0
b	0	0
a	1	1
ba	0	0
bb	0	0

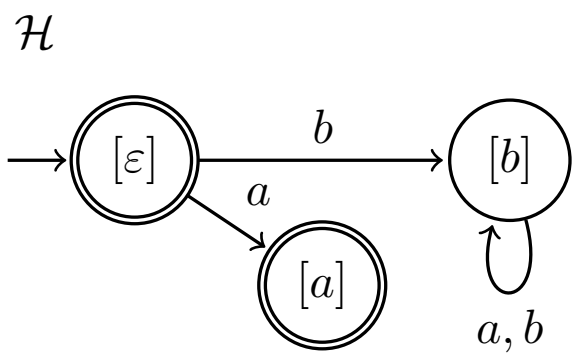


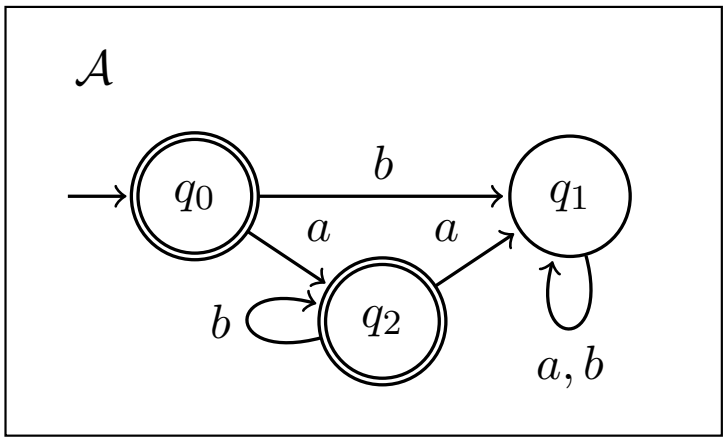


- $\mathcal{H}$ and $\mathcal{A}$ not equivalent!
- Reason: q_0 and q_2 in $\mathcal{A}$ have same future behavior wrt. ε
  - $\Rightarrow$ ε alone not sufficient for distinguishing q_0 (ε) and q_2 (a)
  - $\Rightarrow$ additional columns needed

Refinement

	ε	b
ε	1	0
b	0	0
a	1	1
ba	0	0
bb	0	0

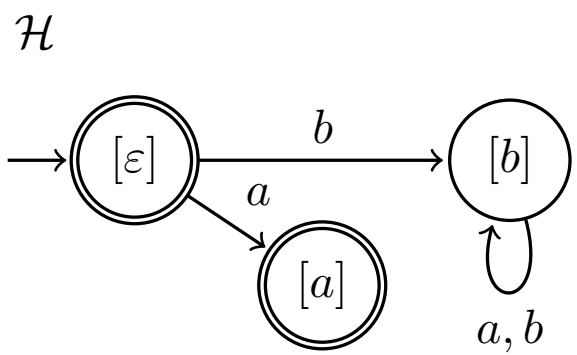


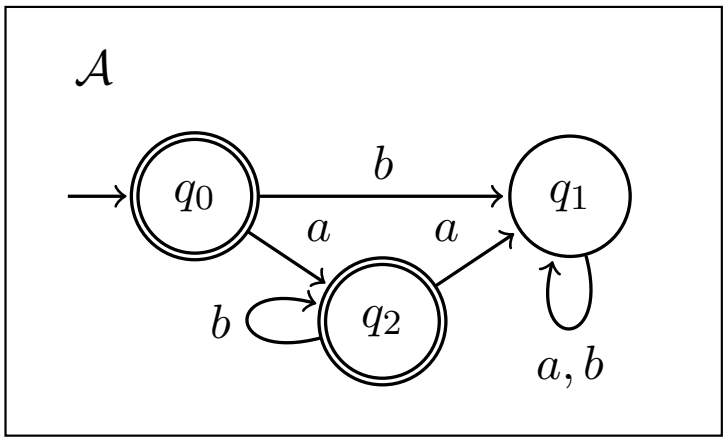


- $\mathcal{H}$ and $\mathcal{A}$ not equivalent!
- Reason: q_0 and q_2 in $\mathcal{A}$ have same future behavior wrt. ε
  - $\Rightarrow$ ε alone not sufficient for distinguishing q_0 (ε) and q_2 (a)
  - $\Rightarrow$ additional columns needed

Refinement

	ε	b
ε	1	0
b	0	0
a	1	1
ba	0	0
bb	0	0
aa	0	0
ab	1	1

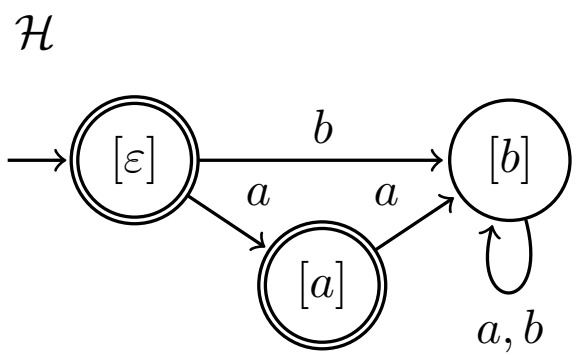


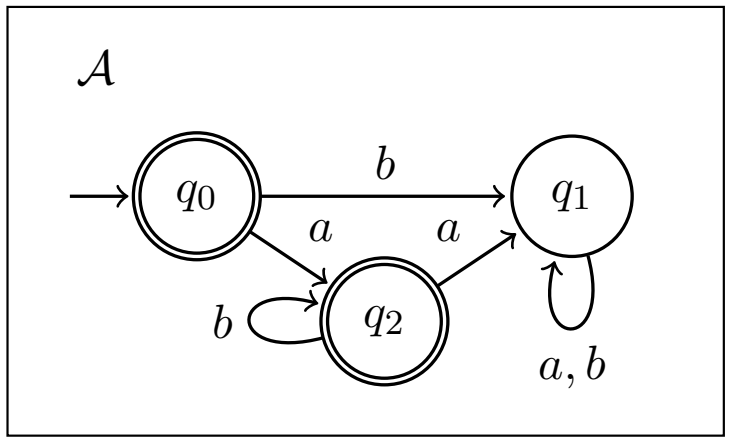


- $\mathcal{H}$ and $\mathcal{A}$ not equivalent!
- Reason: q_0 and q_2 in $\mathcal{A}$ have same future behavior wrt. ε
  - $\Rightarrow$ ε alone not sufficient for distinguishing q_0 (ε) and q_2 (a)
  - $\Rightarrow$ additional columns needed

Refinement

	ε	b
ε	1	0
b	0	0
a	1	1
ba	0	0
bb	0	0
aa	0	0
ab	1	1



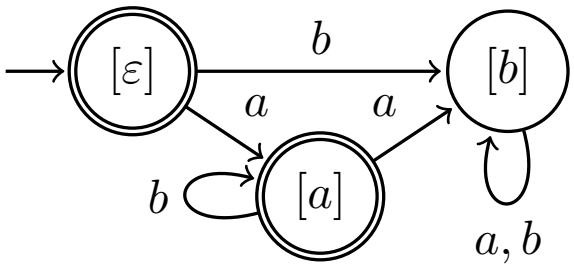


- $\mathcal{H}$ and $\mathcal{A}$ not equivalent!
- Reason: q_0 and q_2 in $\mathcal{A}$ have same future behavior wrt. ε
  - $\Rightarrow$ ε alone not sufficient for distinguishing q_0 (ε) and q_2 (a)
  - $\Rightarrow$ additional columns needed

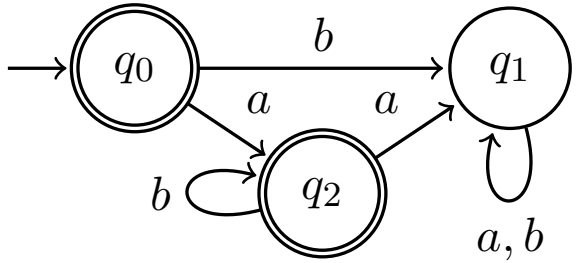
Refinement

	ε	b
ε	1	0
b	0	0
a	1	1
ba	0	0
bb	0	0
aa	0	0
ab	1	1

$\mathcal{H}$



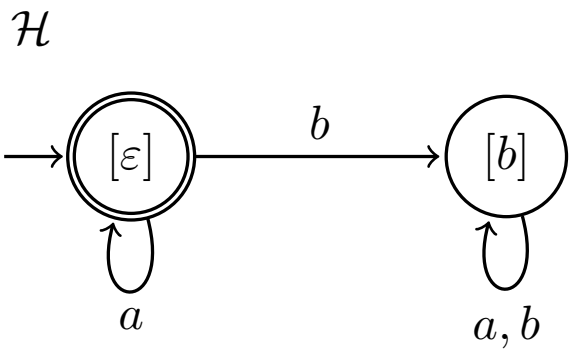
$\mathcal{A}$

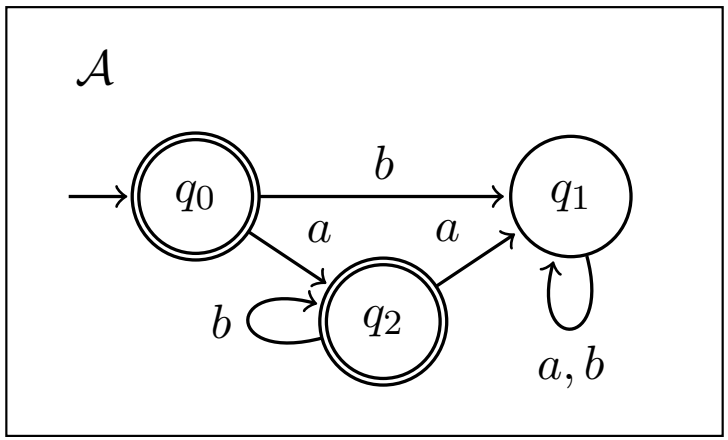


- $\mathcal{H}$ and $\mathcal{A}$ not equivalent!
- Reason: q_0 and q_2 in $\mathcal{A}$ have same future behavior wrt. ε
  - $\Rightarrow$ ε alone not sufficient for distinguishing q_0 (ε) and q_2 (a)
  - $\Rightarrow$ additional columns needed

Refinement: Counterexamples

Reminder: original situation





- 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. $\lambda_{\mathcal{H}}(w) \neq \lambda(w)$

  $\Rightarrow$ i.e., $\mathcal{H}$ predicts wrong output for w
- Example: $w = ab$. $\lambda(ab) = 1$, $\lambda_{\mathcal{H}}(ab) = 0$

Counterexample Decomposition

Counterexample Decomposition

If w is a counterexample wrt. $\mathcal{H}$, then w has a suffix av s.t. for two access sequences $u, u' \in \mathcal{U}$ 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'

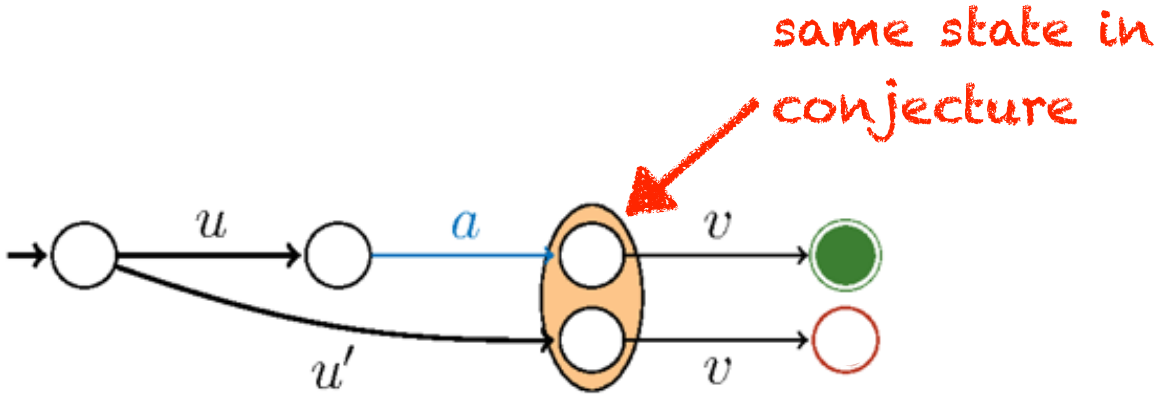
$\Rightarrow$ ua gets moved to upper part

Counterexample Decomposition: 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

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

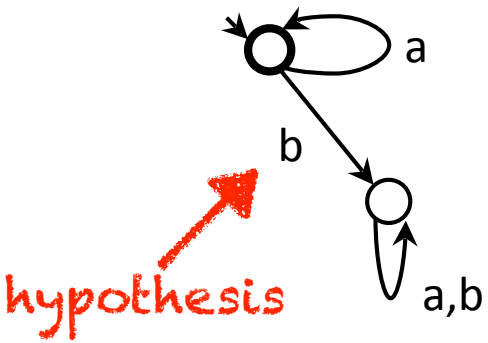


Counterexample Decomposition: Visualization

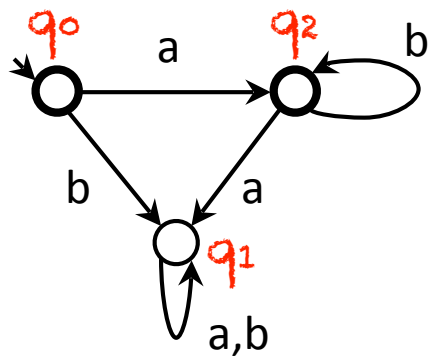
Counterexample Decomposition

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

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



Counterexample: ab

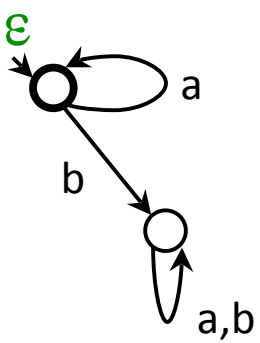


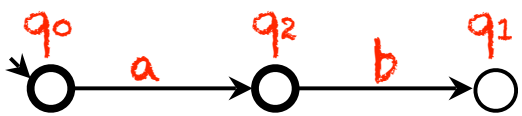
Counterexample Decomposition: Visualization

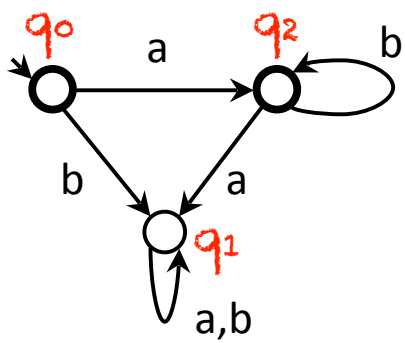
Counterexample Decomposition

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

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





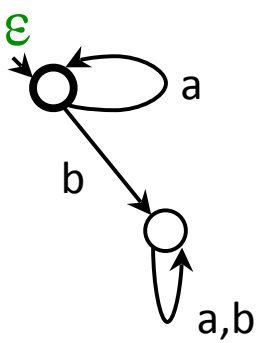


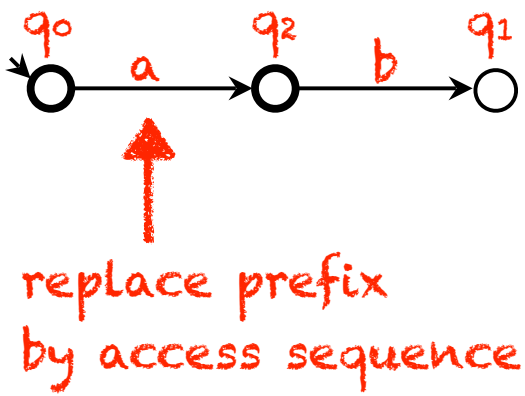
Counterexample Decomposition: Visualization

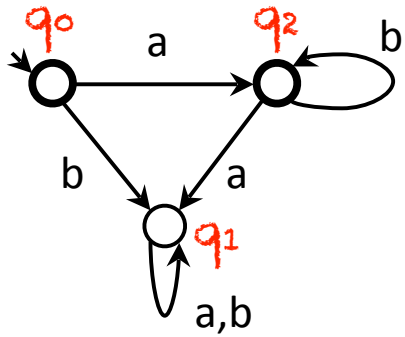
Counterexample Decomposition

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

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





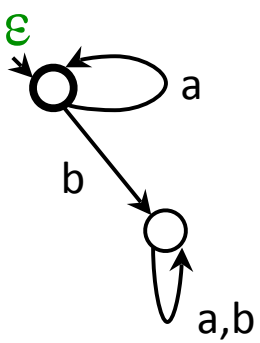


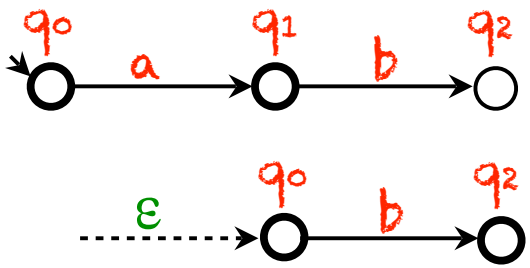
Counterexample Decomposition: Visualization

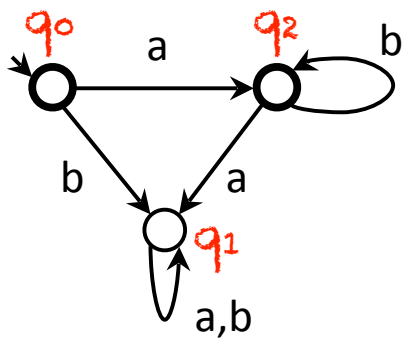
Counterexample Decomposition

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

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





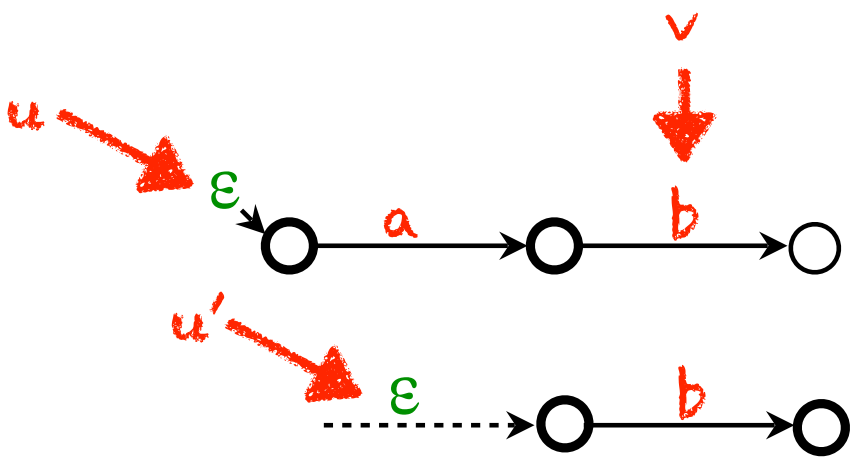


Counterexample Decomposition: Visualization

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. $\mathcal{H}$, 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

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

- 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

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 $u, u' \in \mathcal{U}$ such that ua and u' reach the same state in $\mathcal{H}$, we have

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

- 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) $\mathcal{H}[w] \hat{=}$ state in $\mathcal{H}$ reached by $w = w_1 \dots w_m$
- Access sequence of $q \in Q^{\mathcal{H}}$: unique element $u \in \mathcal{U}$ representing state q (furthermore: $\mathcal{H}[u] = q$)
- $\lfloor w \rfloor_{\mathcal{H}} \hat{=}$ access sequence of state $\mathcal{H}[w]$
- $\pi_{\mathcal{H}}(w, i) =_{df} \lfloor w_1 \dots w_i \rfloor_{\mathcal{H}} w_{i+1} \dots w_m$ $(i \in \{0, \dots, m\})$: 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)
  - $\pi_{\mathcal{H}}(w, m) = \lfloor w \rfloor_{\mathcal{H}} \in \mathcal{U} \Rightarrow \lambda(\pi_{\mathcal{H}}(w, m)) = \lambda_{\mathcal{H}}(\pi_{\mathcal{H}}(w, m))$ ($\lambda(\pi_{\mathcal{H}}(w, m) \cdot \varepsilon)$ determines acceptance of state represented by $\pi_{\mathcal{H}}(w, m) \in \mathcal{U}$)

Binary Search (2)

- $\pi_{\mathcal{H}}(w, i) =_{df} \lfloor w_1 \dots w_i \rfloor_{\mathcal{H}} w_{i+1} \dots w_m$ $(i \in \{0, \dots, m\})$: 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)
  - $\pi_{\mathcal{H}}(w, m) = \lfloor w \rfloor_{\mathcal{H}} \in \mathcal{U} \Rightarrow \lambda(\pi_{\mathcal{H}}(w, m)) = \lambda_{\mathcal{H}}(\pi_{\mathcal{H}}(w, m))$ ($\lambda(\pi_{\mathcal{H}}(w, m) \cdot \varepsilon)$ determines acceptance of state represented by $\pi_{\mathcal{H}}(w, m) \in \mathcal{U}$)
- Obviously: $\lambda_{\mathcal{H}}$ invariant under prefix transformations (i.e., $\lambda_{\mathcal{H}}(\pi_{\mathcal{H}}(w, i)) = \lambda_{\mathcal{H}}(w)$ for all i)
- Breakpoint: $i \in \{0, \dots, m-1\}$ s.t. $\lambda(\pi_{\mathcal{H}}(w, i)) \neq \lambda(\pi_{\mathcal{H}}(w, i+1))$
  - exists if w is a counterexample

Breakpoints

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

$$\begin{aligned} & \lambda(\ \lfloor w_1 \ldots w_i \rfloor_{\mathcal{H}}\ w_{i+1}\ \cdot\ w_{i+2} \ldots w_m\ ) \\ \neq\ & \lambda(\ \lfloor w_1 \ldots w_{i+1} \rfloor_{\mathcal{H}}\ \cdot\ w_{i+2} \ldots w_m\ ) \end{aligned}$$

Breakpoints

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

$$\begin{aligned} & \lambda(\overbrace{\lfloor w_1 \ldots w_i \rfloor_{\mathcal{H}}}^{u} \overbrace{w_{i+1}}^{a} \cdot \overbrace{w_{i+2} \ldots w_m}^{v}) \\ \neq\ & \lambda(\underbrace{\lfloor w_1 \ldots w_{i+1} \rfloor_{\mathcal{H}}}_{u'} \cdot \underbrace{w_{i+2} \ldots w_m}_{v}) \end{aligned}$$

  - with $u =_{df} \lfloor w_1 \ldots w_i \rfloor_{\mathcal{H}}$, $a =_{df} w_{i+1}$, $v =_{df} w_{i+2} \ldots w_m$:
    $\Rightarrow$ decomposition theorem

Breakpoints

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

$$\begin{aligned} & \lambda(\overbrace{\lfloor w_1 \ldots w_i \rfloor_{\mathcal{H}}}^{u} \overbrace{w_{i+1}}^{a} \cdot \overbrace{w_{i+2} \ldots w_m}^{v}) \\ \neq\ & \lambda(\underbrace{\lfloor w_1 \ldots w_{i+1} \rfloor_{\mathcal{H}}}_{u'} \cdot \underbrace{w_{i+2} \ldots w_m}_{v}) \end{aligned}$$

  - with $u =_{df} \lfloor w_1 \ldots w_i \rfloor_{\mathcal{H}}$, $a =_{df} w_{i+1}$, $v =_{df} w_{i+2} \ldots w_m$:
    $\Rightarrow$ decomposition theorem

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

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

The Cost of Learning

Common measure: query complexity ($\hat{=}$ 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 $\mathcal{A}$
  - $k \hat{=}$ size of Σ
  - $m \hat{=}$ length of longest counterexamples

The Cost of Learning

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

- Parameters:
  - $n \hat{=}$ number of states of $\mathcal{A}$
  - $k \hat{=}$ size of Σ
  - $m \hat{=}$ 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 ($\hat{=}$ required number of test cases)

- Parameters:
  - $n \hat{=}$ number of states of $\mathcal{A}$
  - $k \hat{=}$ size of Σ
  - $m \hat{=}$ 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
- 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 $\mathcal{O}(kn^2)$

The Cost of Learning (2)

Size of observation table: $\mathcal{O}(kn^2)$

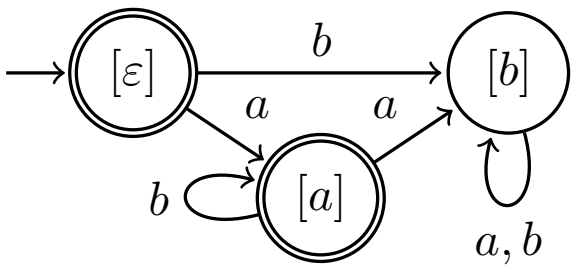
- Plus: counterexample analysis (Rivest&Schapire)
  - Binary search: $\mathcal{O}(\log m)$ queries per counterexample
  - at most $n-1$ counterexamples $\Rightarrow$ total $\mathcal{O}(n \log m)$ queries for counterexample analysis
- Total query complexity: $\mathcal{O}(kn^2 + n \log m)$
- Known lower bound [Balcázar *et al.*, 1997]: $\Omega(kn^2)$

Discrimination Trees

Redundancies in Observation Table

	ε	b
ε	1	0
b	0	0
a	1	1
ba	0	0
bb	0	0
aa	0	0
ab	1	1

$\mathcal{H}$

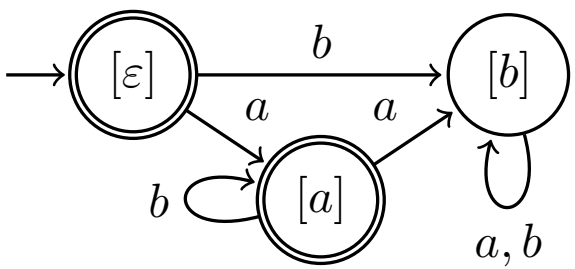


Discrimination Trees

Redundancies in Observation Table

	ε	b
ε	1	0
b	0	0
a	1	1
ba	0	0
bb	0	0
aa	0	0
ab	1	1

$\mathcal{H}$



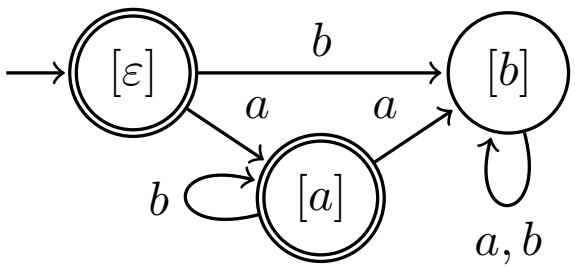
- Value 0 in ε-column sufficient to distinguish $[b]$ from other two states

Discrimination Trees

Redundancies in Observation Table

	ε	b
ε	1	0
b	0	
a	1	1
ba	0	
bb	0	
aa	0	
ab	1	1

$\mathcal{H}$

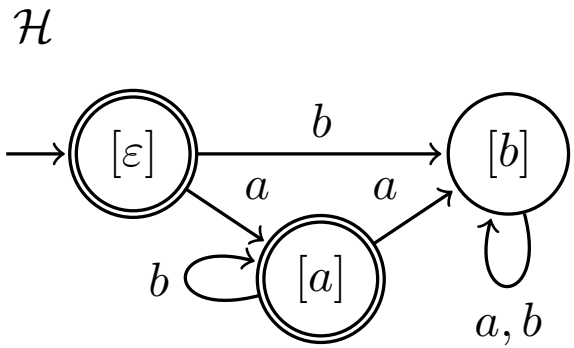


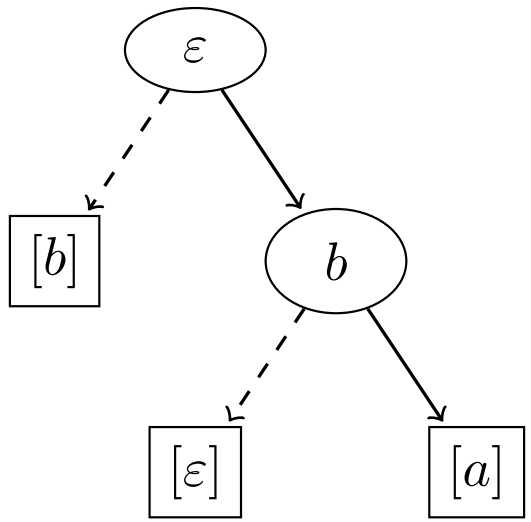
- Value 0 in ε-column sufficient to distinguish $[b]$ from other two states

Discrimination Trees

Redundancies in Observation Table

	ε	b
ε	1	0
b	0	
a	1	1
ba	0	
bb	0	
aa	0	
ab	1	1



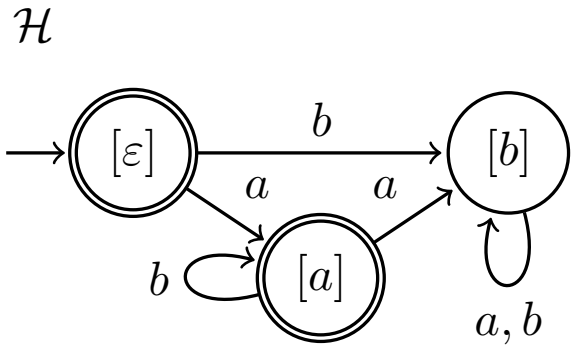


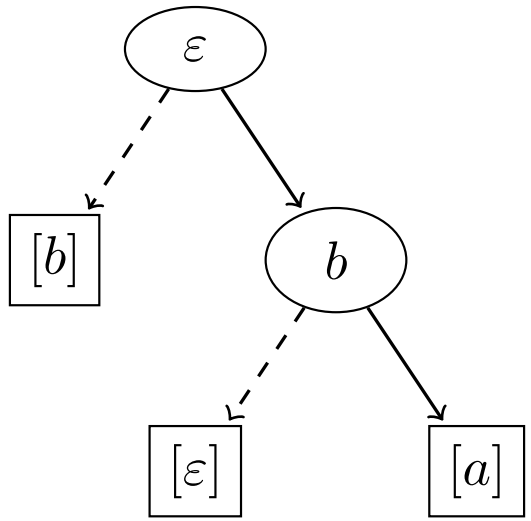
- Value 0 in ε-column sufficient to distinguish $[b]$ from other two states

Discrimination Trees

Redundancies in Observation Table

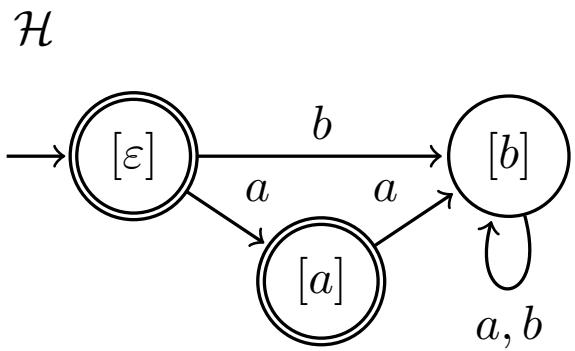
	ε	b
ε	1	0
b	0	
a	1	1
ba	0	
bb	0	
aa	0	
ab	1	1

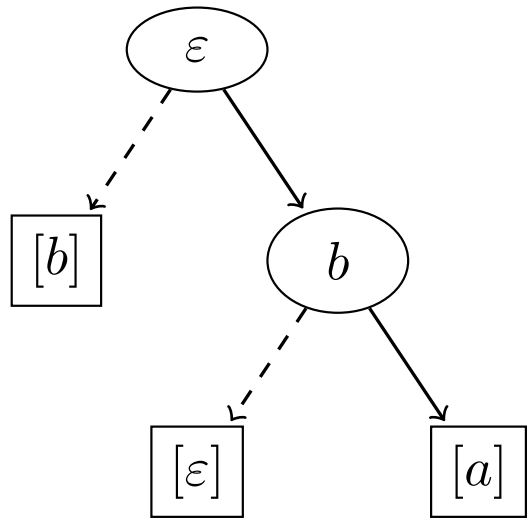




- Value 0 in ε-column sufficient to distinguish $[b]$ from other two states
- Discrimination Tree redundancy-free: exactly one distinguishing suffix for each pair of states (LCA)

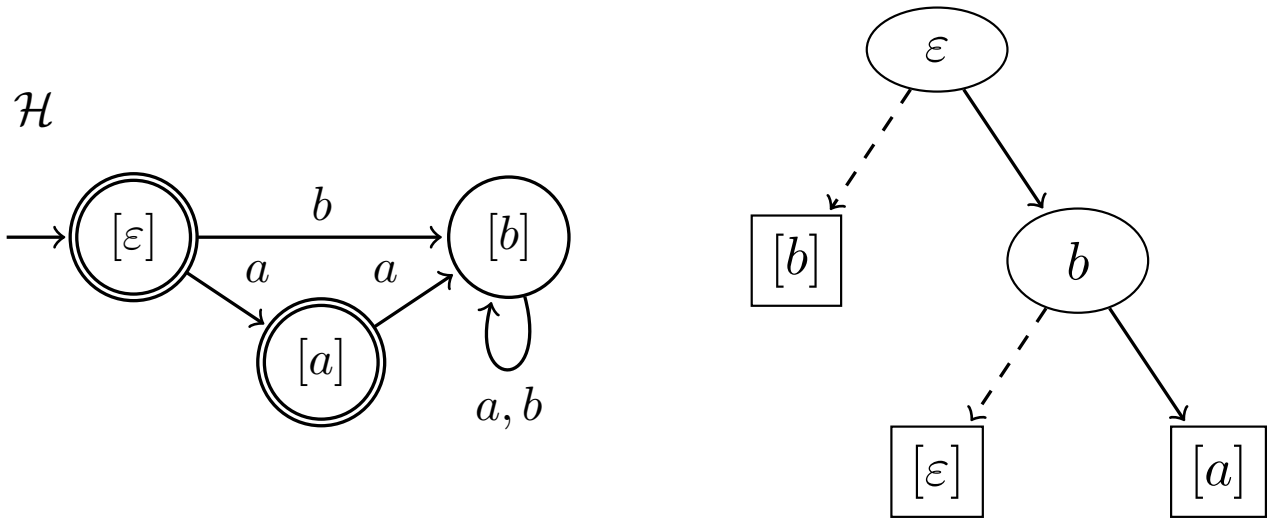
Discrimination Trees: Sifting





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

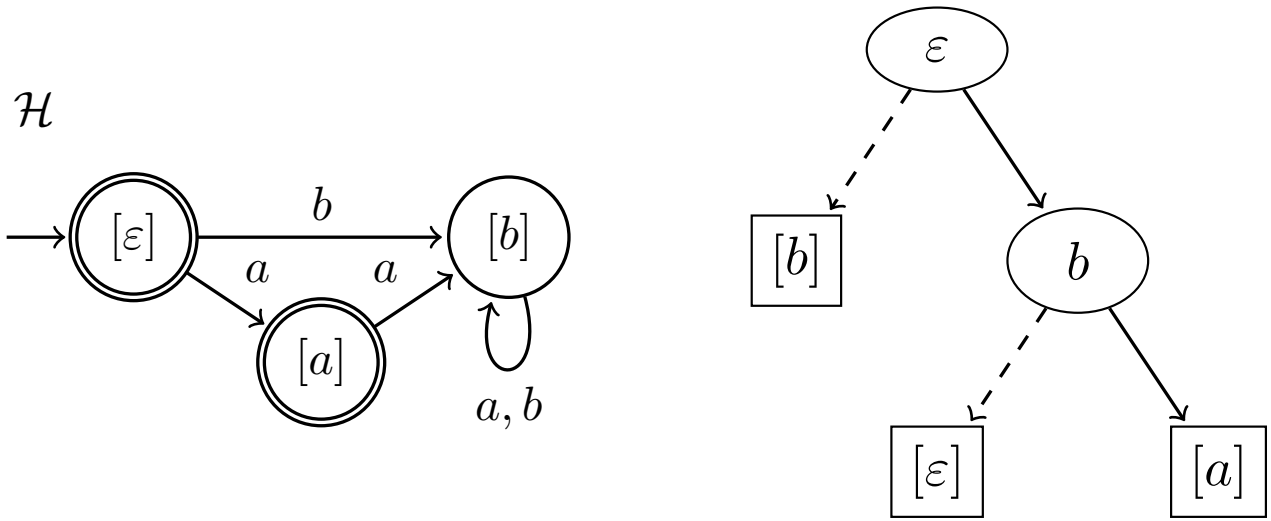
Discrimination Trees: Sifting



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

- Access sequence of transition: access seq. of state + symbol (in this case: ab)

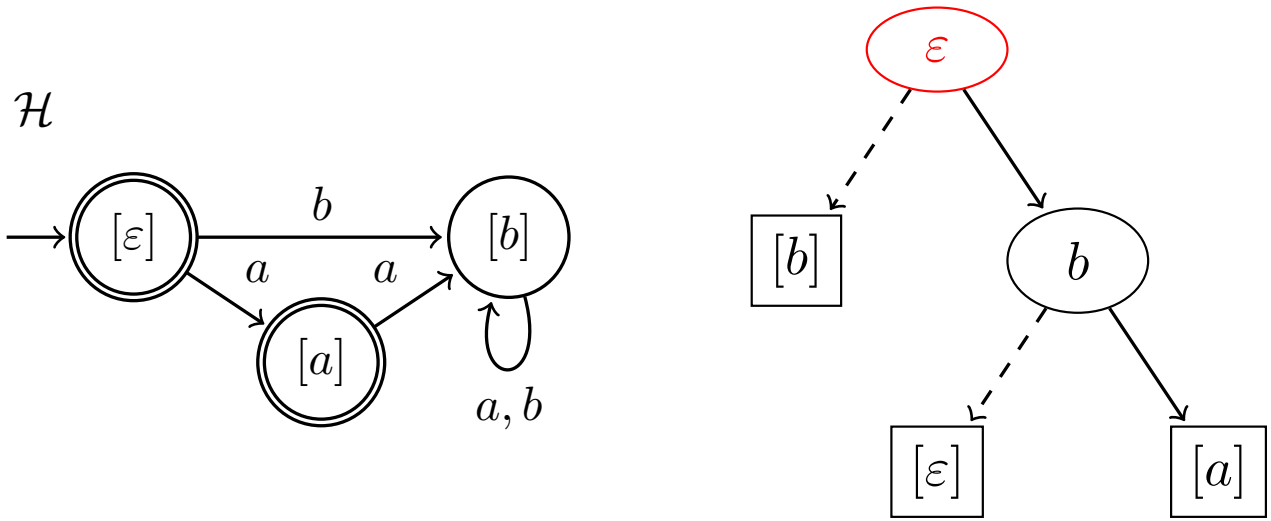
Discrimination Trees: Sifting



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

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

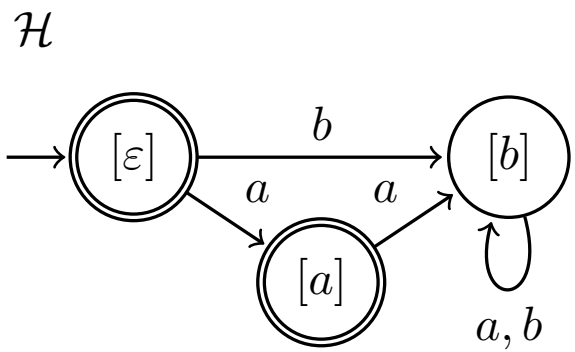
Discrimination Trees: Sifting

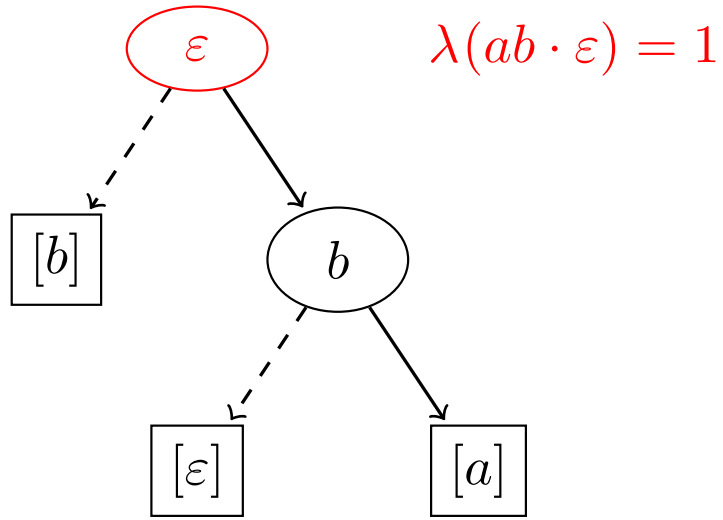


Determining the target state of the b-transition of state $[a]$:

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

Discrimination Trees: Sifting

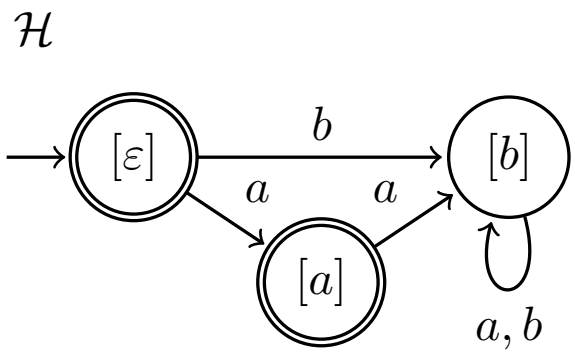


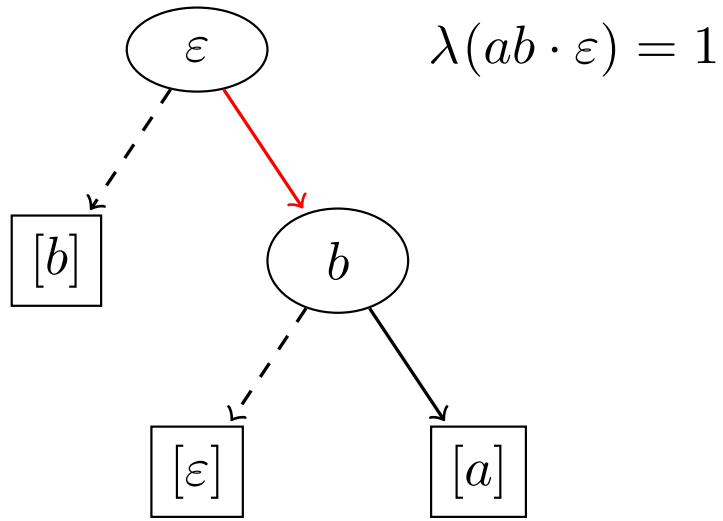


Determining the target state of the b-transition of state $[a]$:

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

Discrimination Trees: Sifting

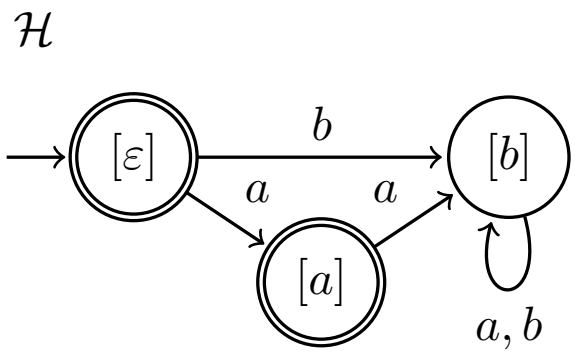


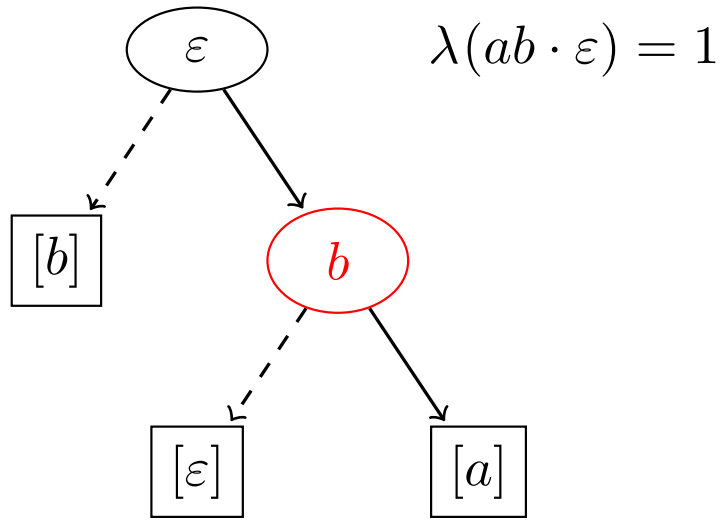


Determining the target state of the b-transition of state $[a]$:

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

Discrimination Trees: Sifting

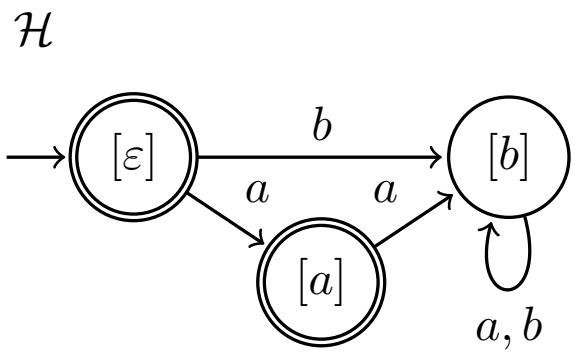


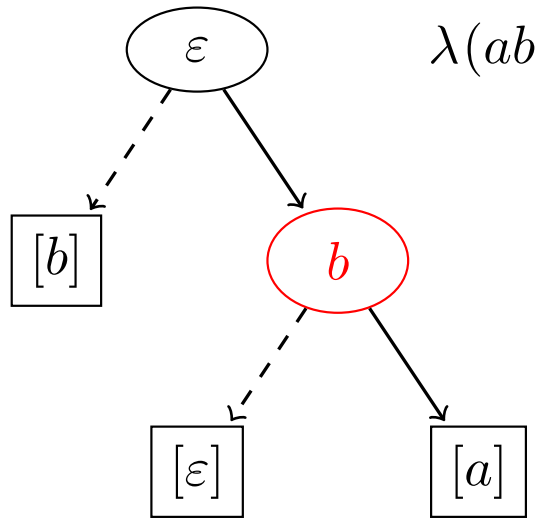


Determining the target state of the b-transition of state $[a]$:

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

Discrimination Trees: Sifting

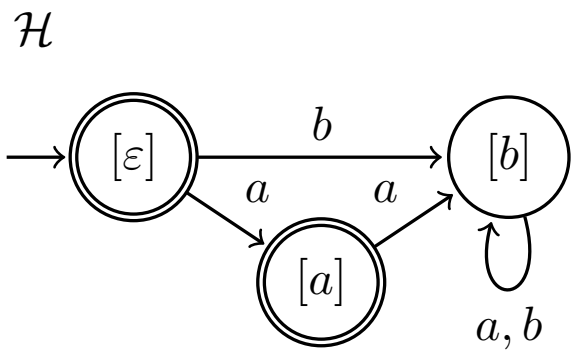


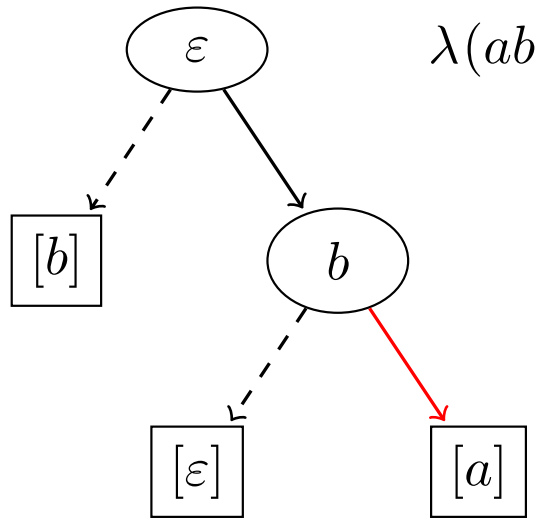


Determining the target state of the b-transition of state $[a]$:

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

Discrimination Trees: Sifting

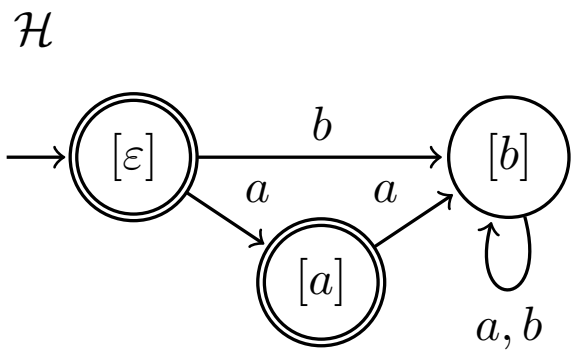


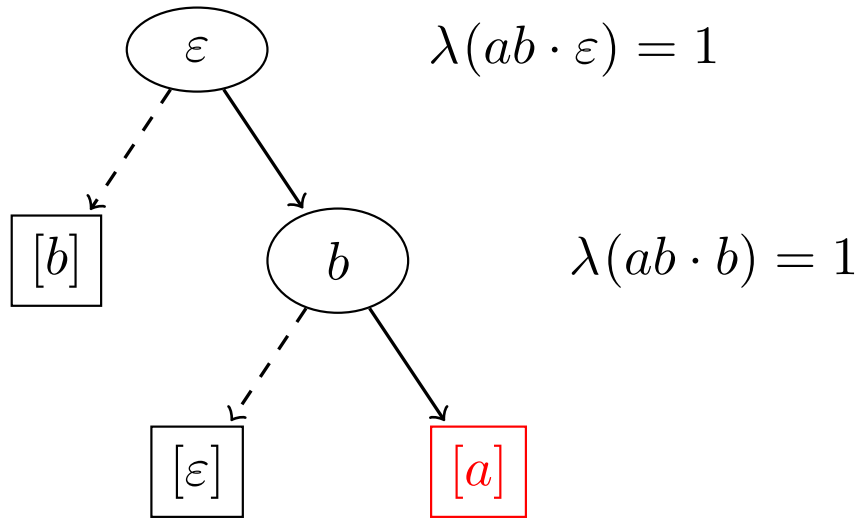


Determining the target state of the b-transition of state $[a]$:

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

Discrimination Trees: Sifting

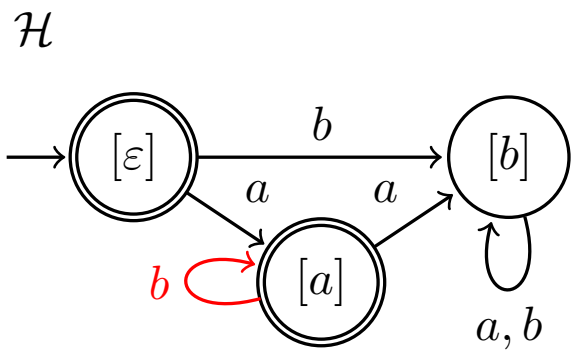


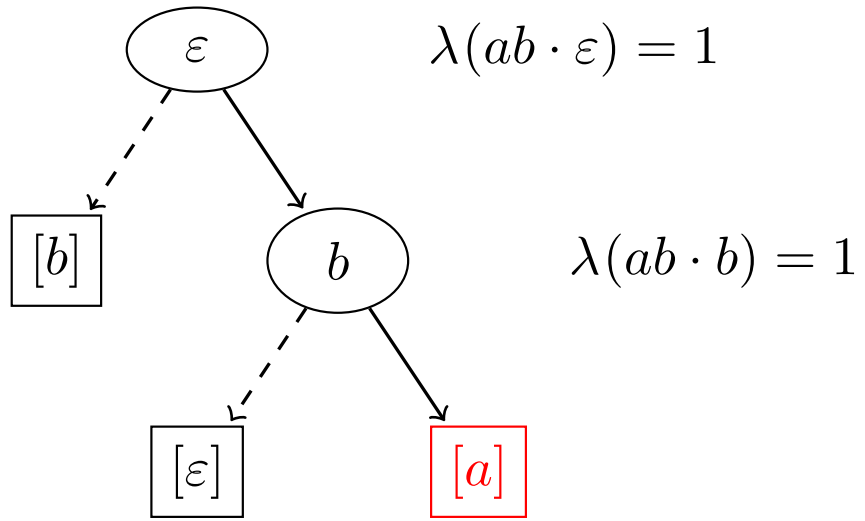


Determining the target state of the b-transition of state $[a]$:

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

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)
- "Sift" ab into discrimination tree

Discrimination Trees: Refinement

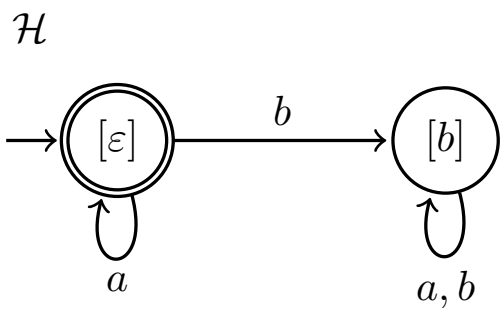
Reminder: Counterexample decomposition yields $u, u' \in \mathcal{U}$, $a \in \Sigma$, $v \in \Sigma^*$

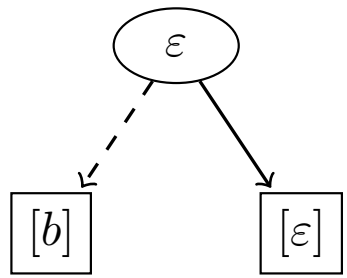
- Observation Table: simply add v to columns
- Does not work for discrimination trees (locality)!

Discrimination Trees: Refinement

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)!



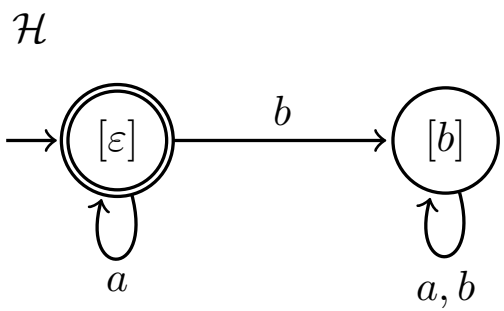


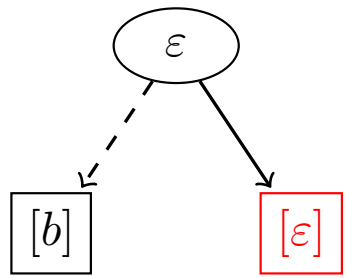
- Refinement: split leaf corresponding to $u' = \varepsilon$

Discrimination Trees: Refinement

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)!



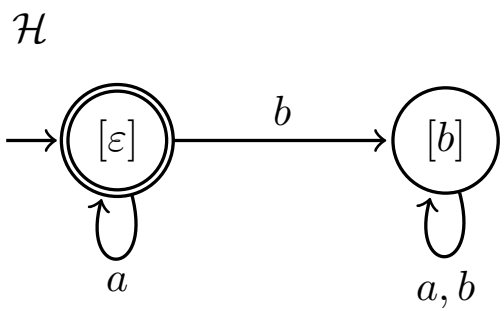


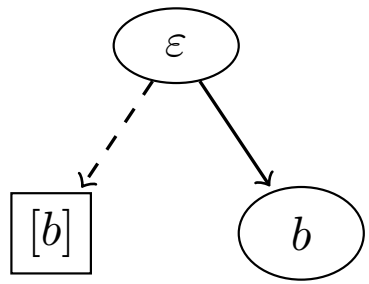
- Refinement: split leaf corresponding to $u' = \varepsilon$

Discrimination Trees: Refinement

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)!



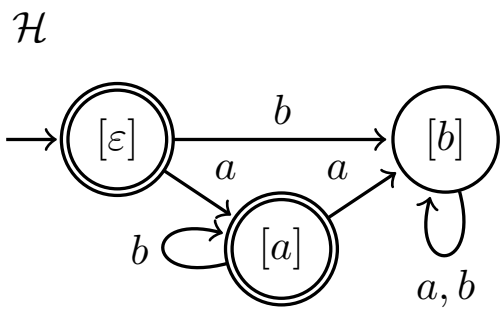


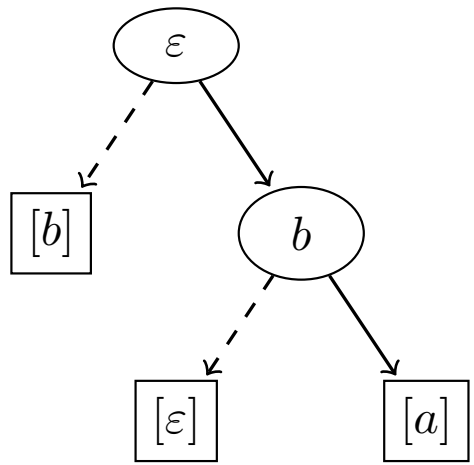
- Refinement: split leaf corresponding to $u' = \varepsilon$
  - Suffix at new inner node: $v = b$

Discrimination Trees: Refinement

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

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

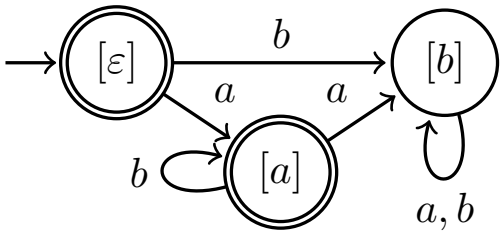




- Refinement: split leaf corresponding to $u' = \varepsilon$
  - Suffix at new inner node: $v = b$
  - New children (leaves): $[u']$ and new state $[ua]$ $(u = \varepsilon)$

Spanning Tree Hypothesis

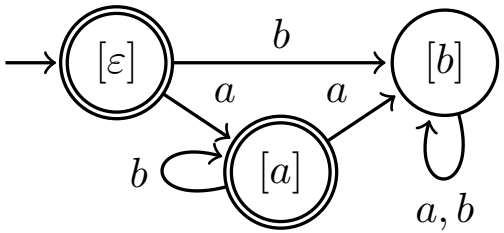
$\mathcal{H}$



Observation: new states derived from transitions

Spanning Tree Hypothesis

$\mathcal{H}$



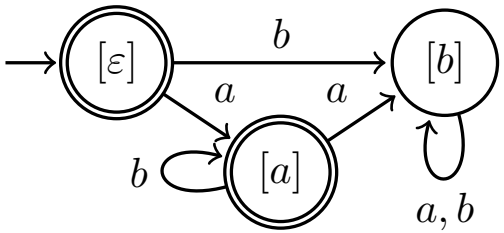
Observation: new states derived from transitions

- 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

$\mathcal{H}$



Observation: new states derived from transitions

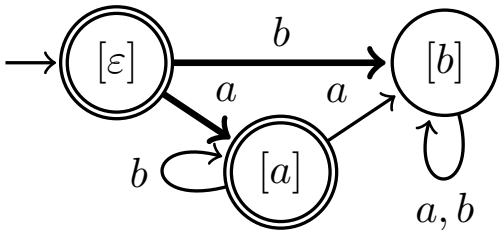
- i.e., access sequence of the form ua, where $u \in \mathcal{U}, a \in \Sigma$

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

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

Spanning Tree Hypothesis

$\mathcal{H}$



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

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

$\Rightarrow$ Spanning tree can be used to represent $\mathcal{U}$

Outline

Practical Challenges

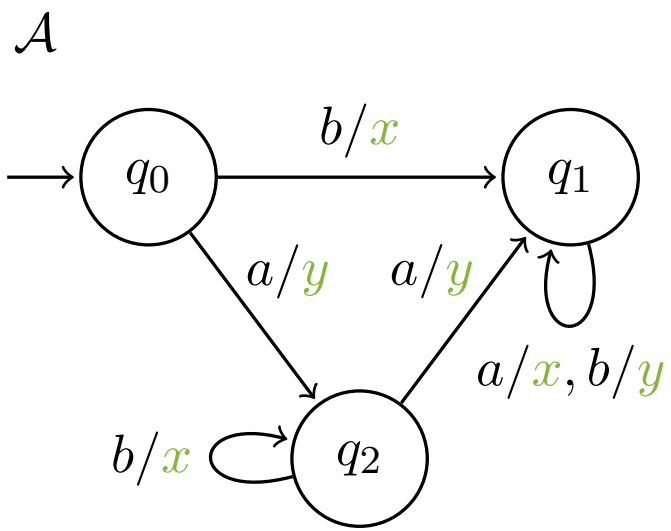
Main practical challenge: Realizing membership and equivalence queries

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

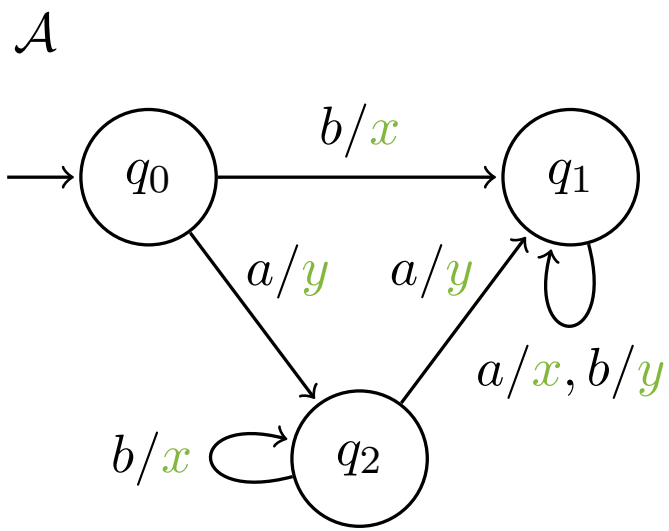
But first: DFAs not adequate for modeling reactive systems

- Better-suited model: Mealy machines

Mealy machines

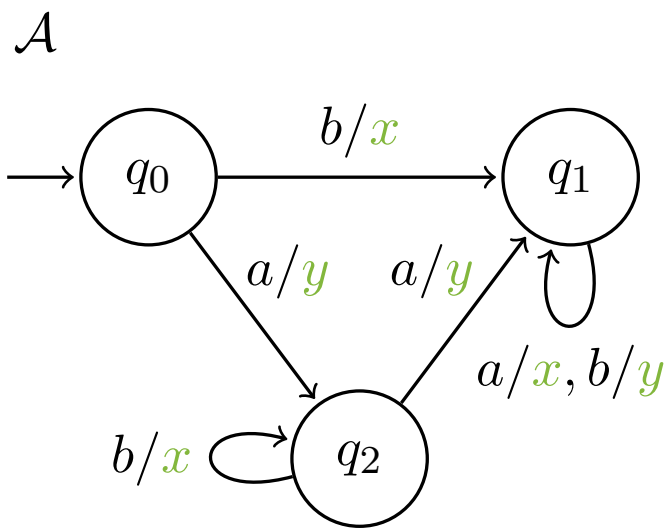


Mealy machines



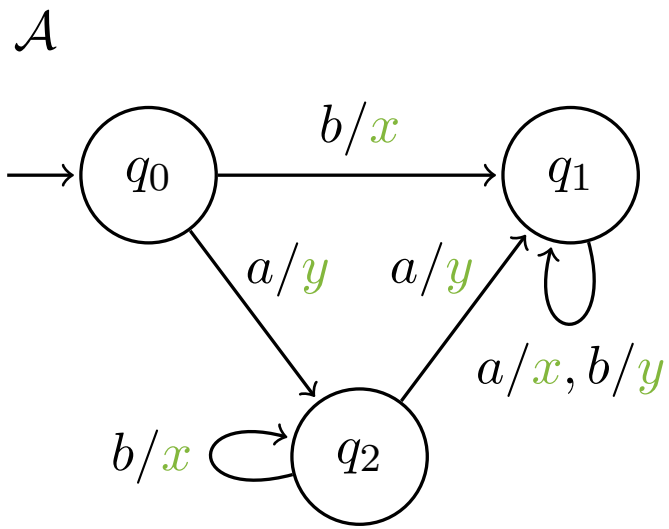
- No accepting/rejecting states, but transition outputs

Mealy machines



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

Mealy machines



- No accepting/rejecting states, but transition outputs
- Output function: $\lambda : \Sigma^* \to \Omega^*$ (Ω: 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 Σ
  - e.g., API calls, communication primitives etc.
- Step 2: define semantics for each symbol

Realizing Membership Queries

- Membership Queries $\hat{=}$ test cases
- Step 1: define learning alphabet Σ
  - 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 Σ
  - 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.)

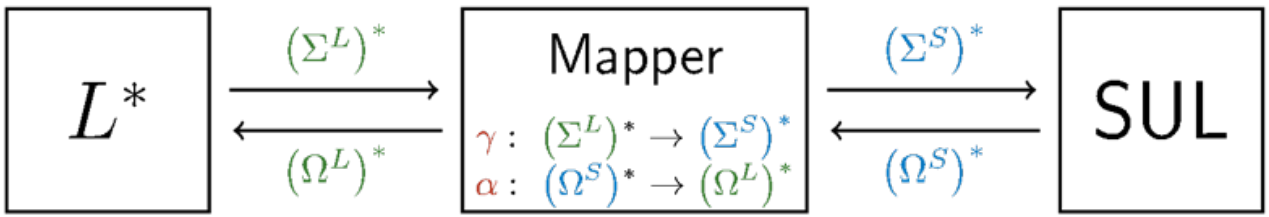
Realizing Membership Queries

- Membership Queries $\hat{=}$ test cases
- Step 1: define learning alphabet Σ
  - 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

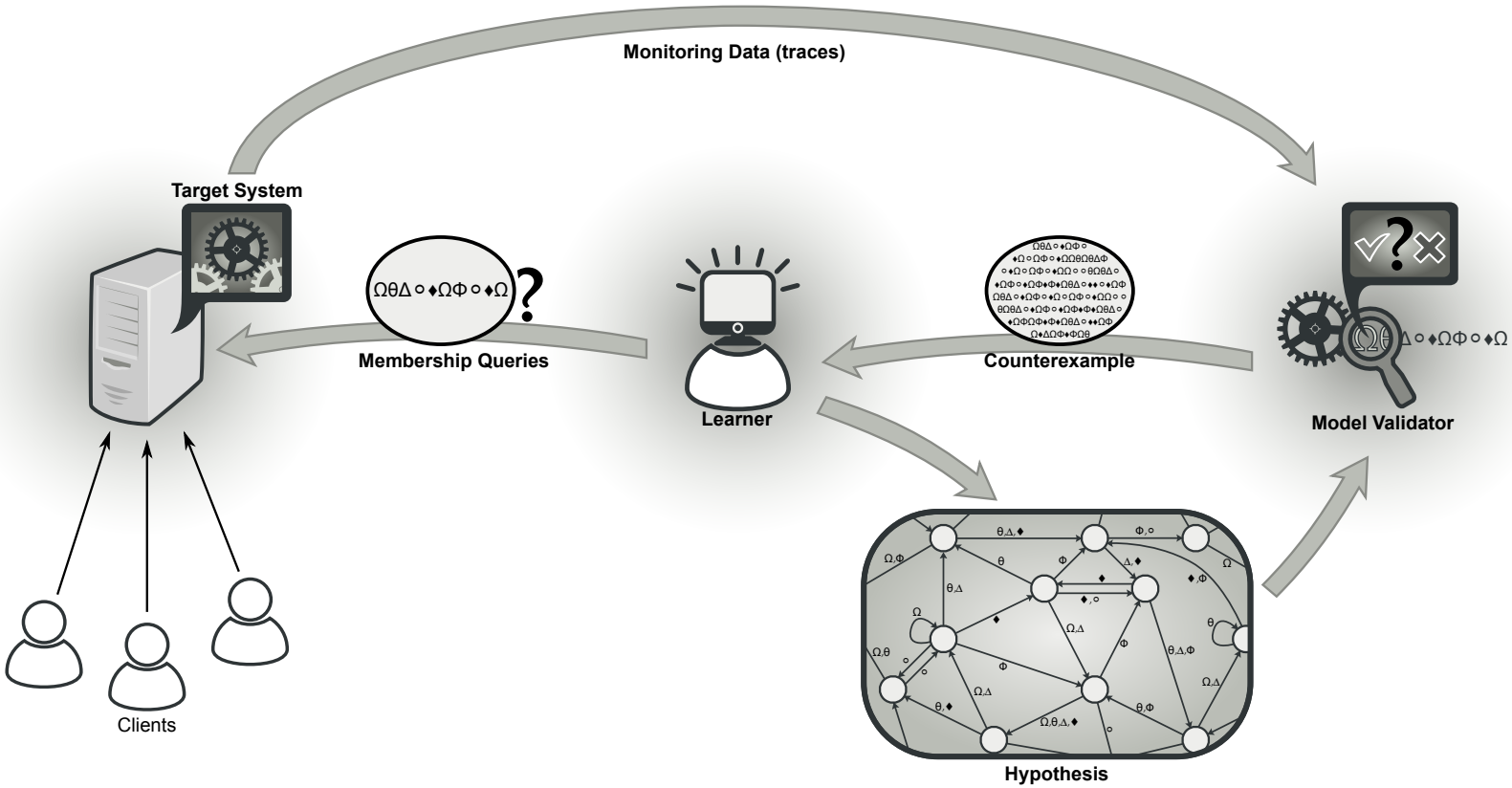
- Classical approach: conformance testing methods
  - W-method, Wp-method, UIO method etc.
  - relative guarantees (e.g., assuming the target system has at most Δn additional states) at the cost of an exponential number of membership queries

Approximating Equivalence Queries

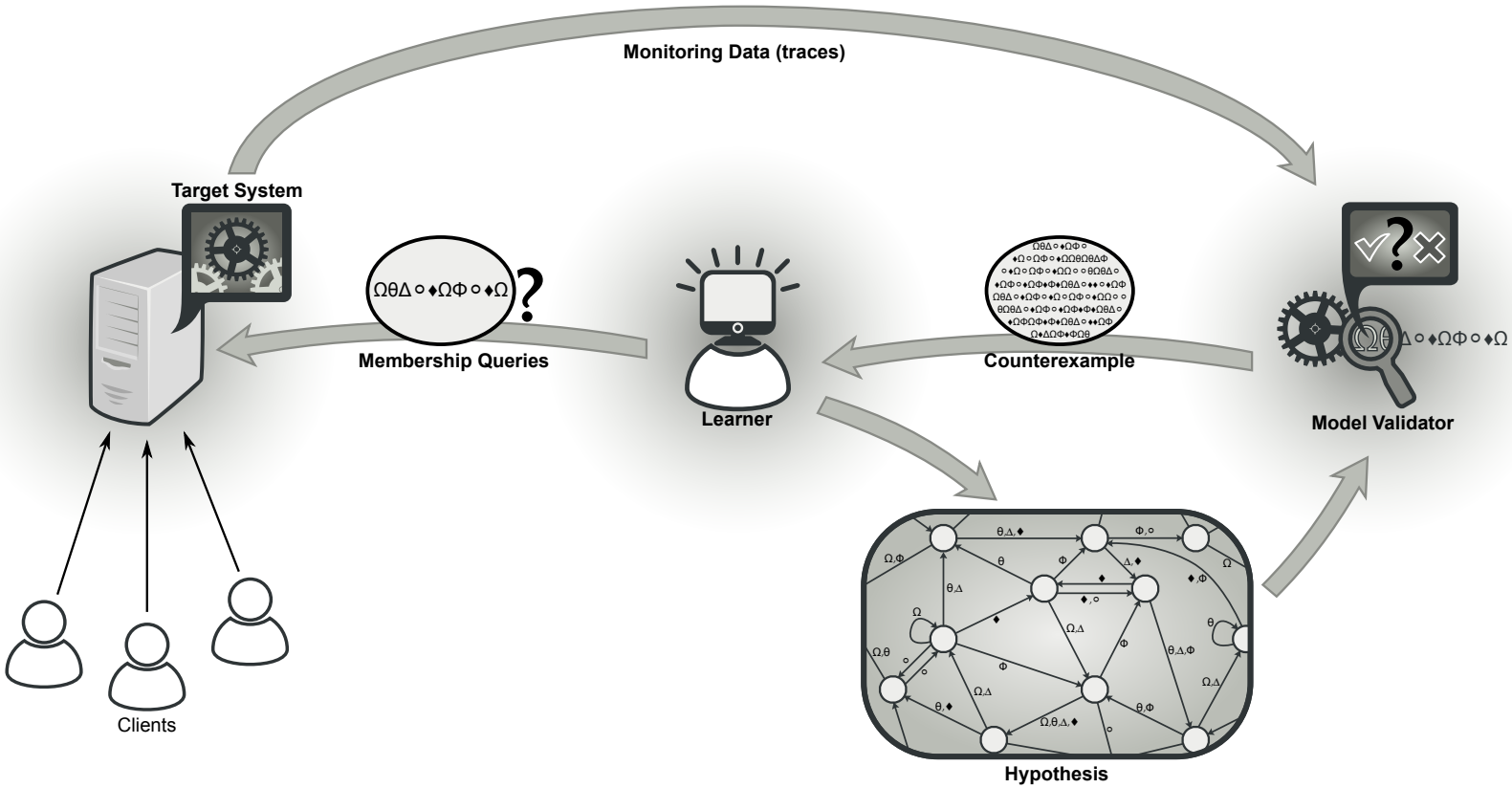
- Classical approach: conformance testing methods
  - W-method, Wp-method, UIO method etc.
  - relative guarantees (e.g., assuming the target system has at most Δn additional states) at the cost of an exponential number of membership queries
- Often more viable: log-file analysis or monitoring

  $\Rightarrow$ "life-long learning"

Combining Learning and Monitoring

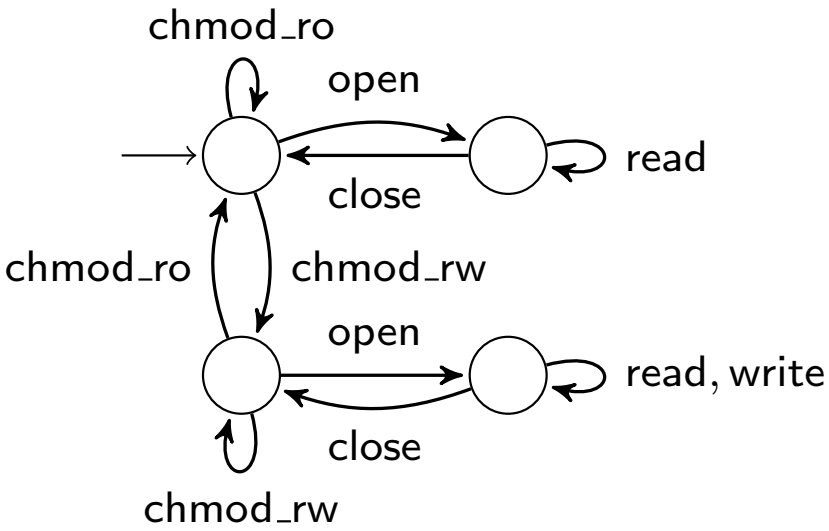


Combining Learning and Monitoring

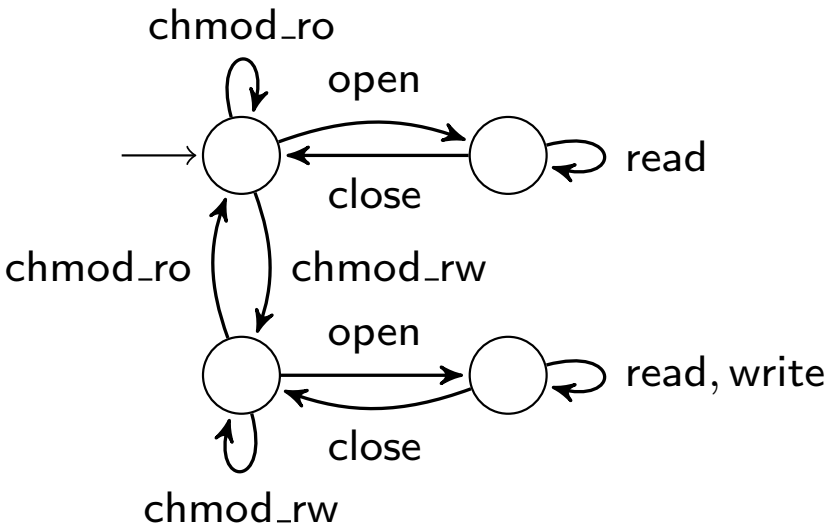


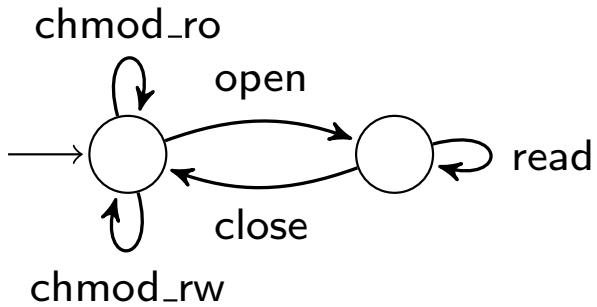
- Executions may run for a long time ⇒ long counterexamples

Counterexamples through Monitoring:

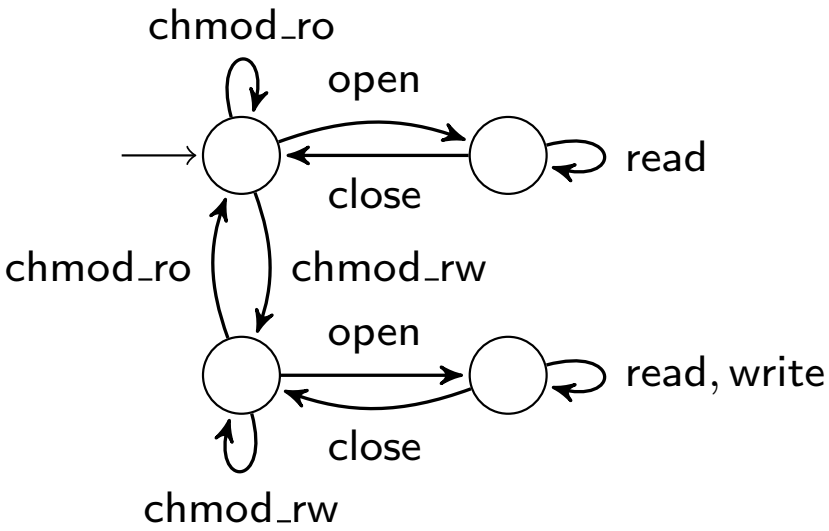


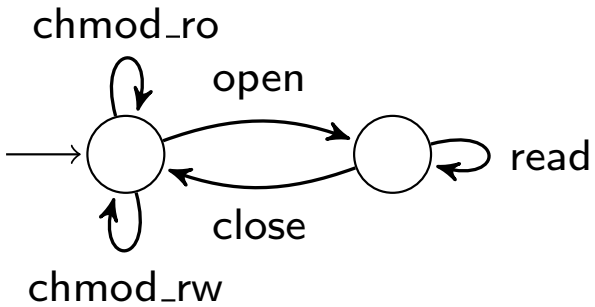
Counterexamples through Monitoring:





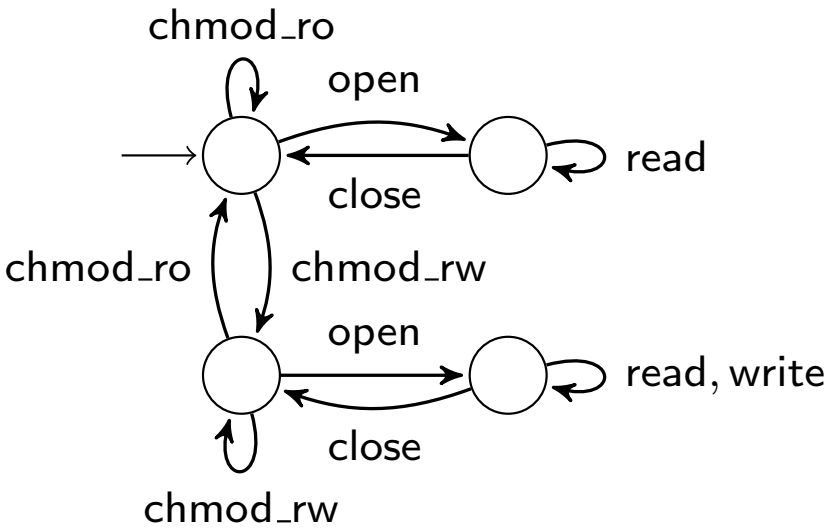
Counterexamples through Monitoring:

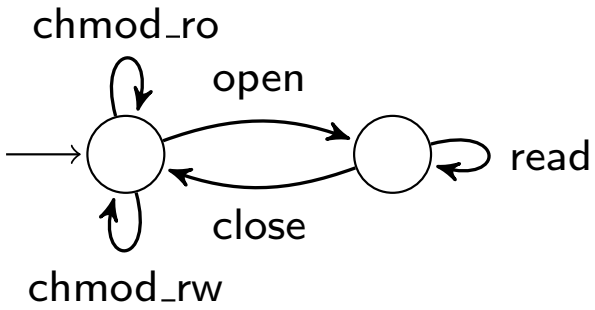




$$\overbrace{\text{open read close open read close chmod\_rw}}^{prefix}\ \underbrace{\text{open read close open read close open write}}_{suffix}$$

Counterexamples through Monitoring:





$$\overbrace{\text{open read close open read close chmod\_rw}}^{prefix}\ \underbrace{\text{open read close open read close open write}}_{suffix}$$

- "Optimal" counterexample chmod_rw open write

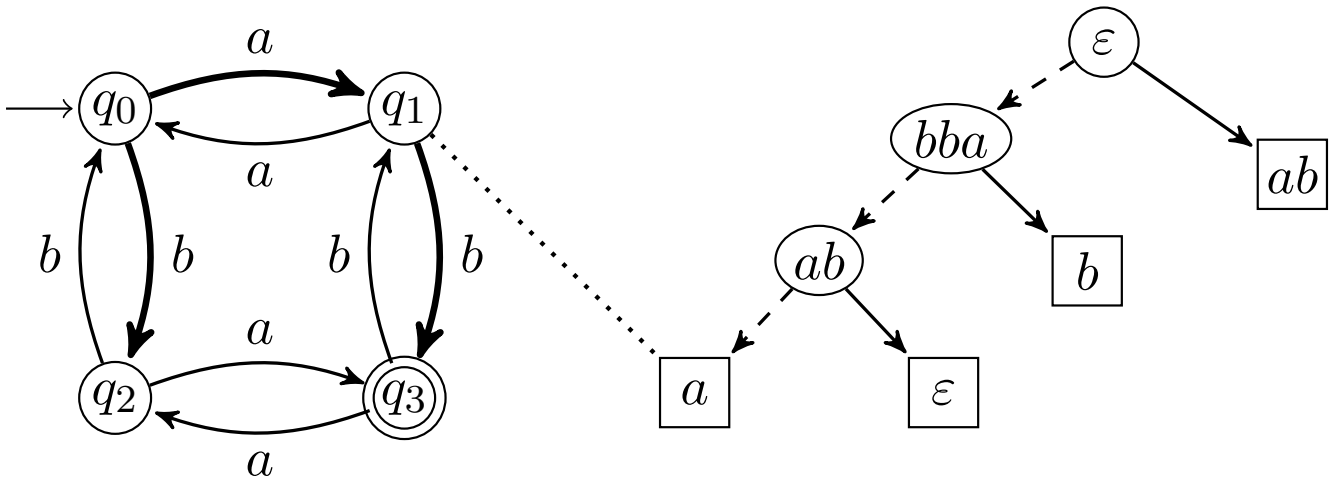
The Problem of Long Counterexamples

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

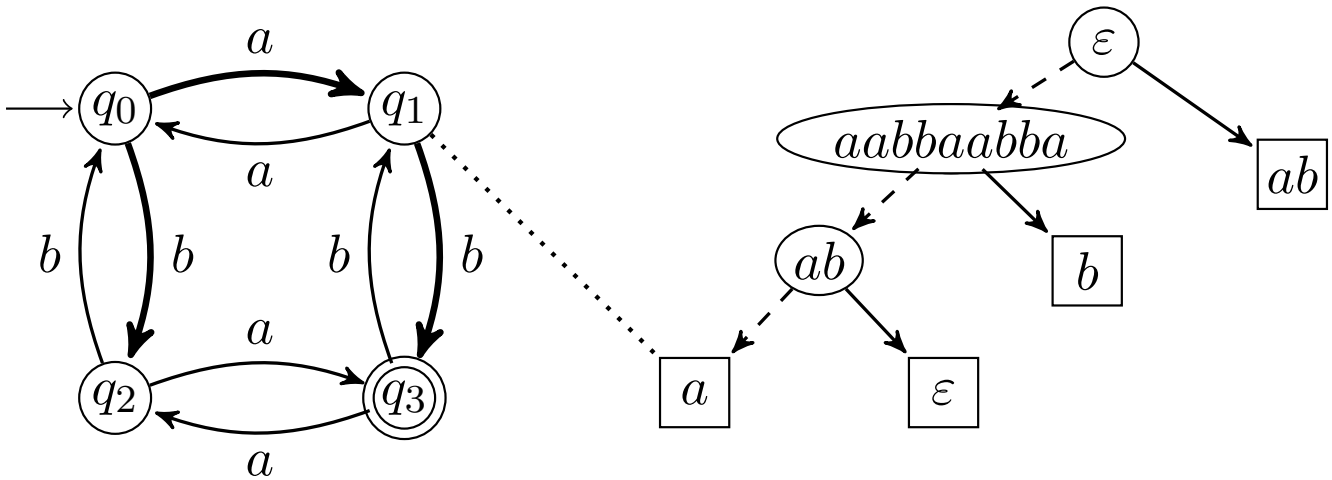
$\Rightarrow$ length of suffix v only bounded by m (length of counterexample)

So what is the problem with long suffixes?

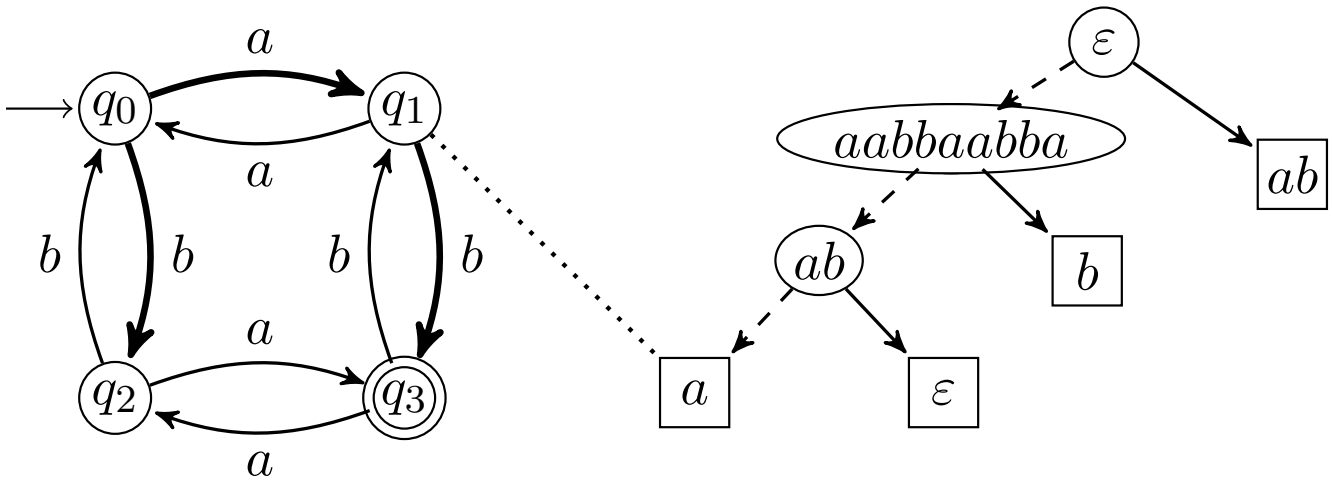
The Problem of Long Suffixes



The Problem of Long Suffixes

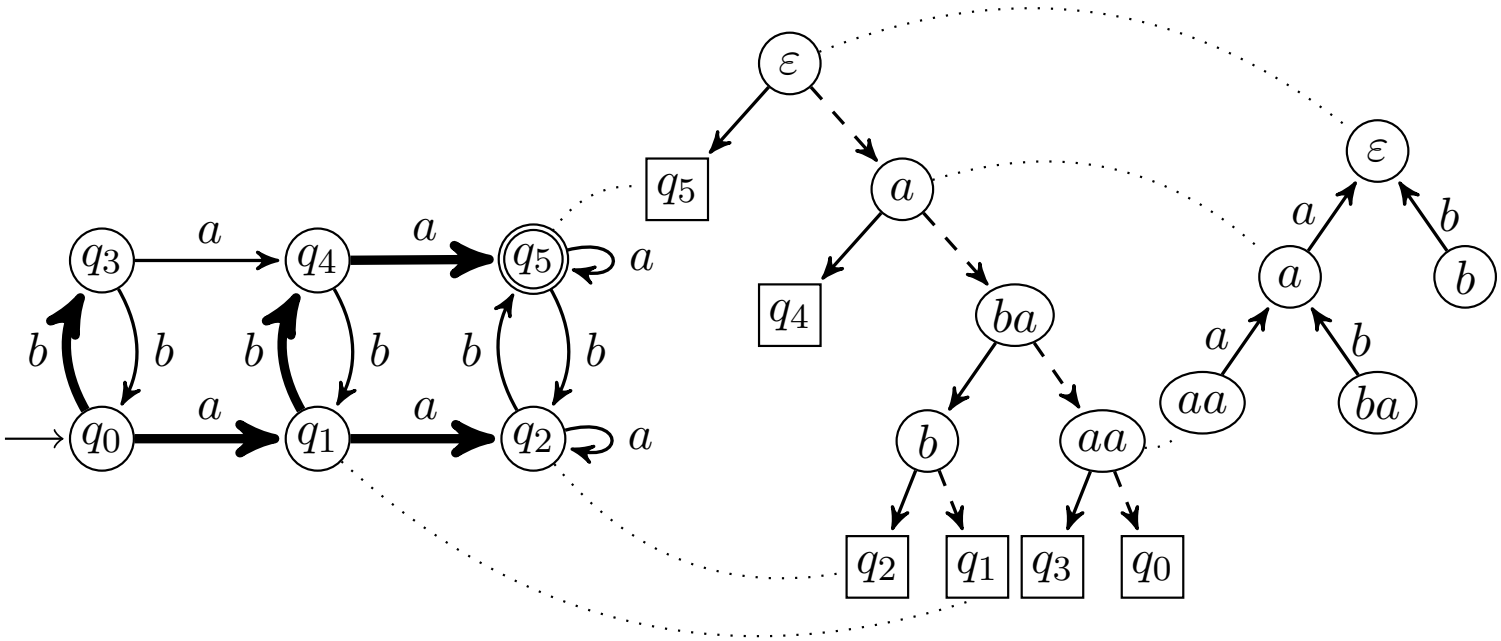


The Problem of Long Suffixes



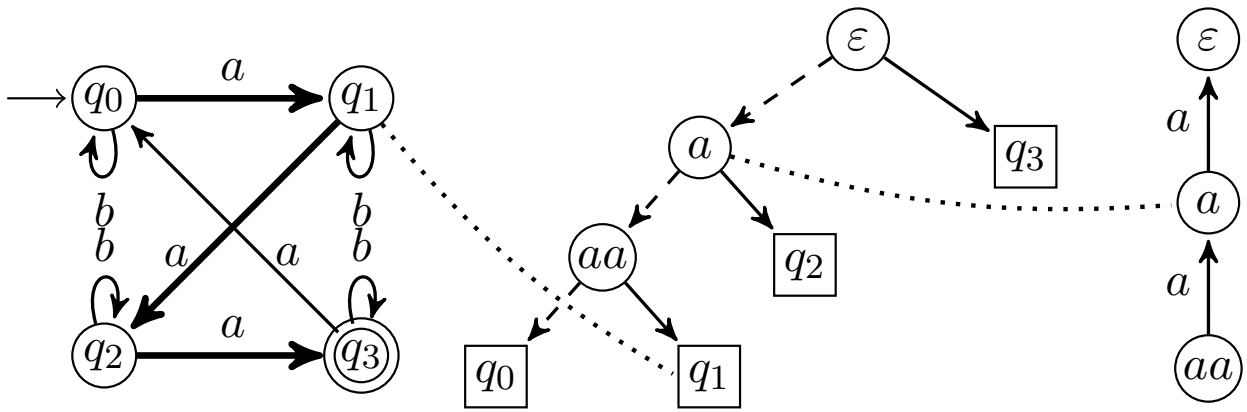
- Consequence: long membership queries
  - $aa\mathit{aabbaabba}, bb\mathit{aabbaabba}, \ldots$
  - Typically: time required for performing membership query linear in its length
- Even worse: impact not "local" but persists throughout entire process

The TTT Algorithm

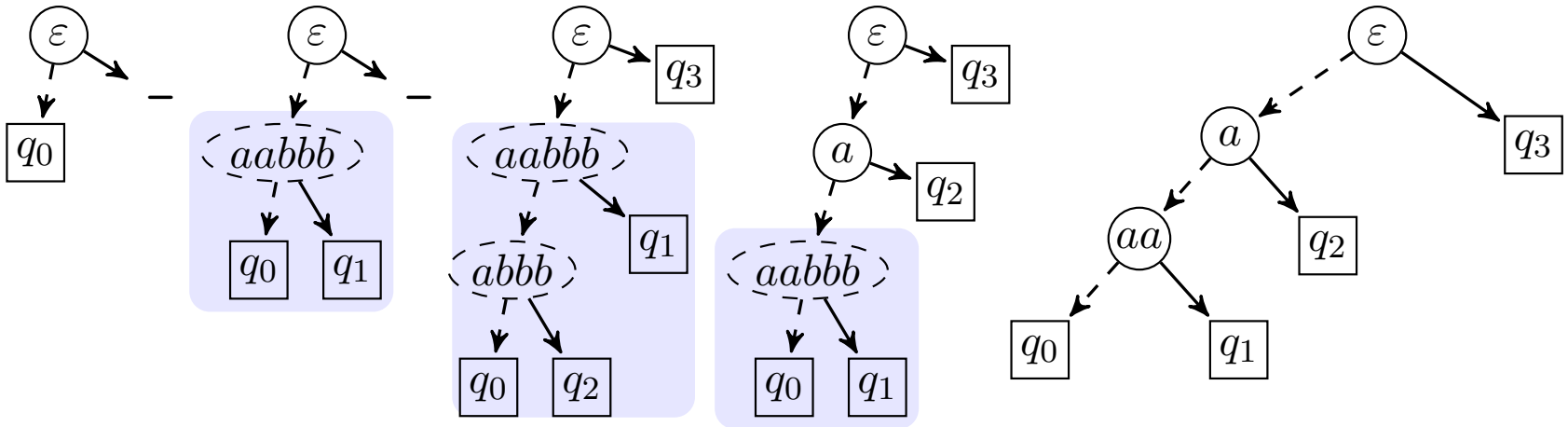


- Idea: "normalize" suffixes to establish suffix-closedness
  - Length of suffixes then bounded by n, organized in Trie
- Spanning **T**ree, Discrimination **T**ree, Suffix **T**rie
  - Linear (optimal) space complexity

TTT – Another Example

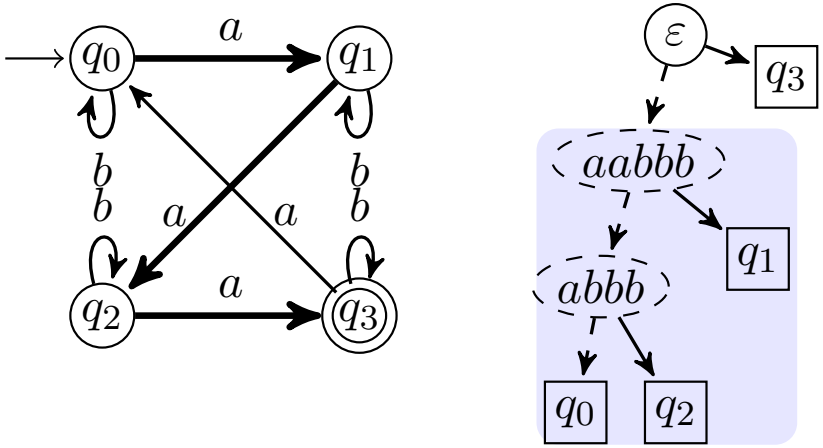


How does it work?



- 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

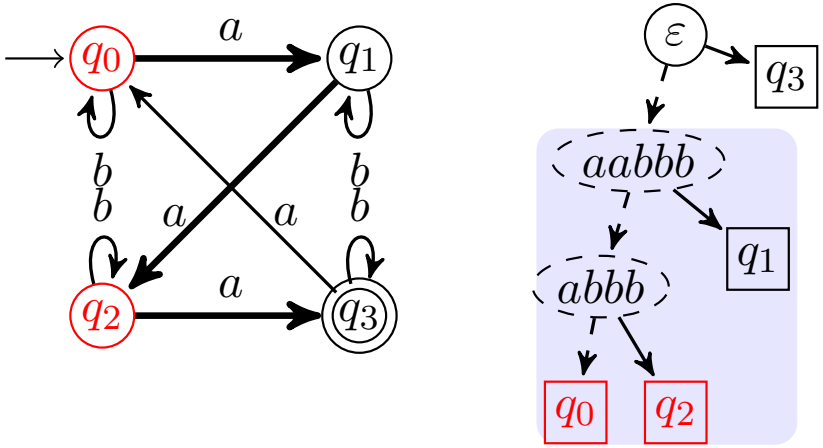
Discriminator Finalization – Idea



Find:

- 2 states q, q' in the same block

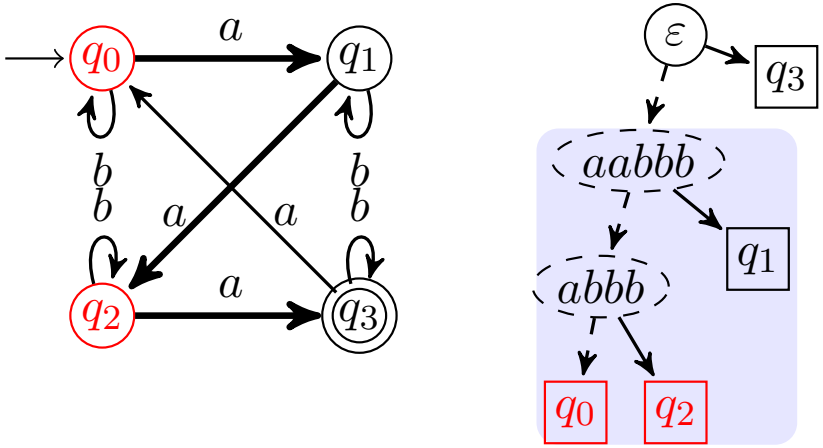
Discriminator Finalization – Idea



Find:

- 2 states q, q' in the same block

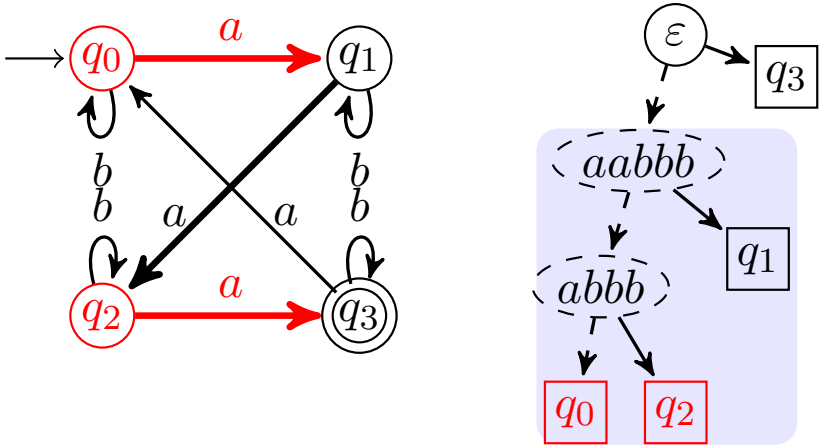
Discriminator Finalization – Idea



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

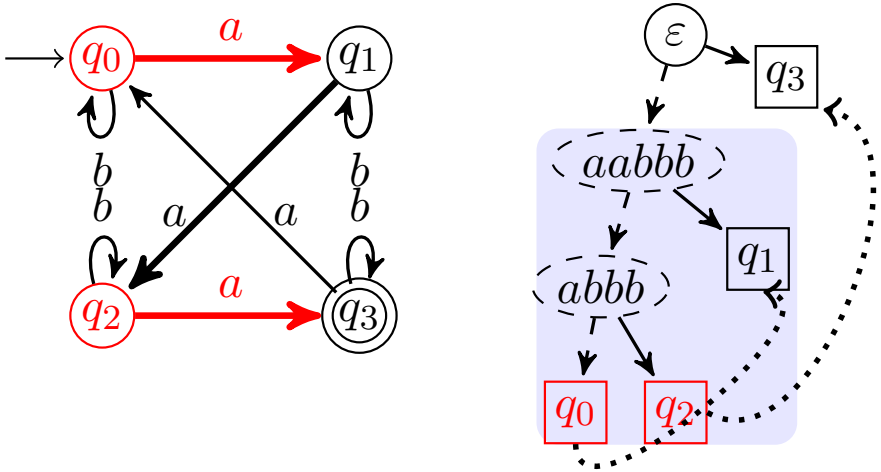
Discriminator Finalization – Idea



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

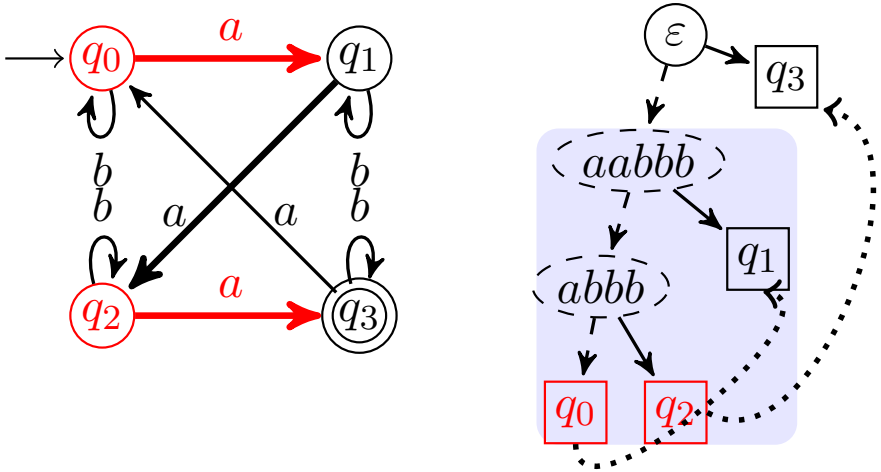
Discriminator Finalization – Idea



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

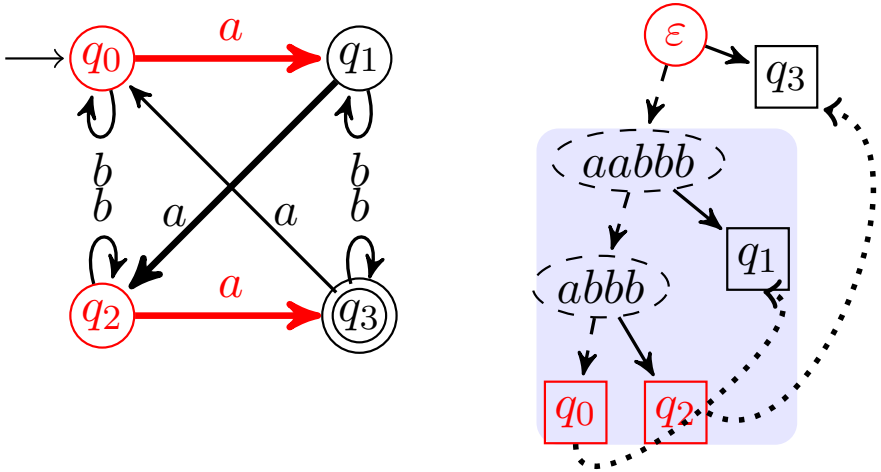
Discriminator Finalization – Idea



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

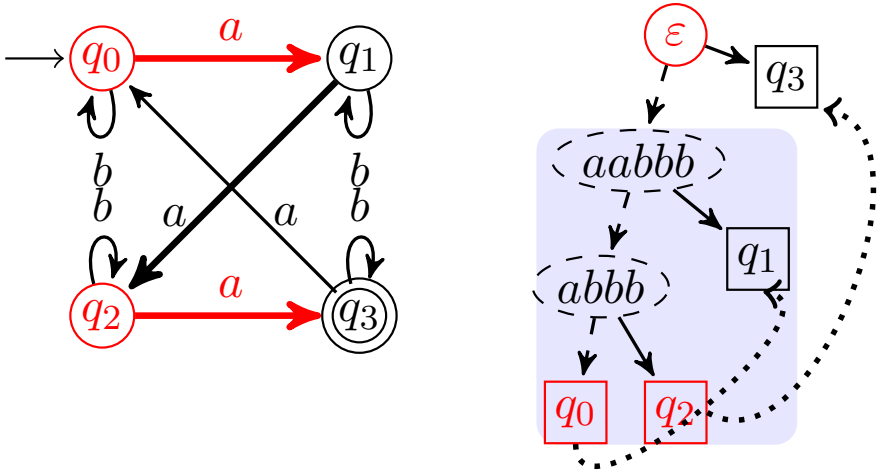
Discriminator Finalization – Idea



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

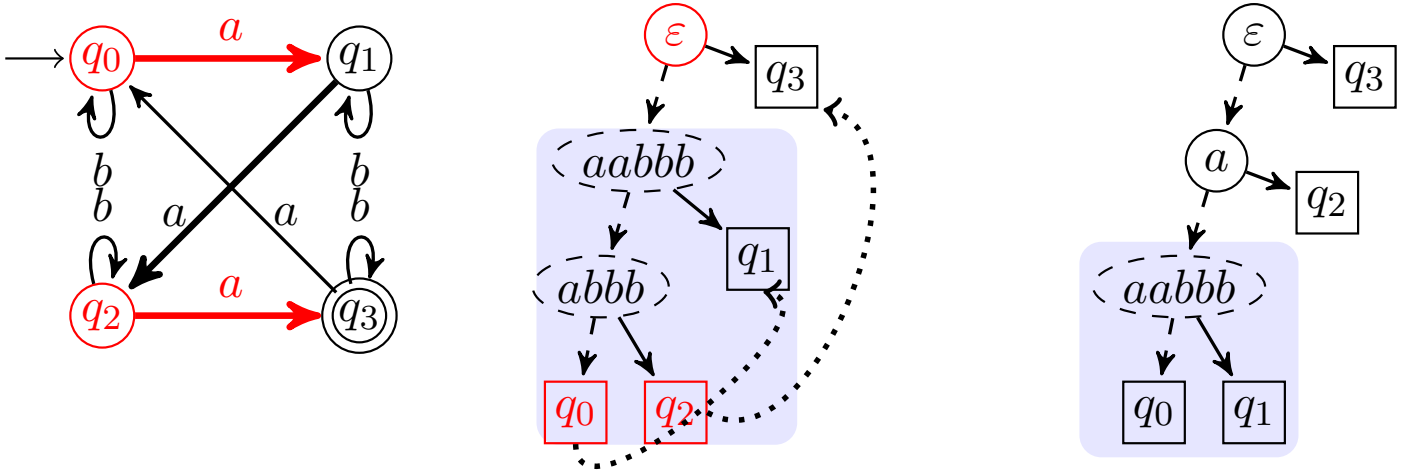
Discriminator Finalization – Idea



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
  - av separates q and $q' \Rightarrow$ new final suffix

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
  - av separates q and $q' \Rightarrow$ new final suffix

Discriminator Finalization – Correctness

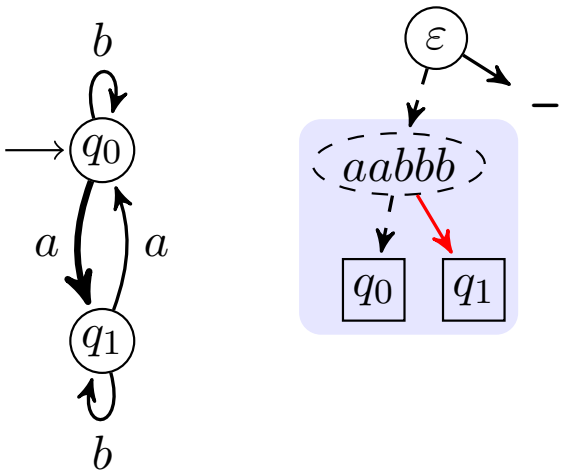
Q: Can we always find such q, q', a?

Discriminator Finalization – Correctness

Q: Can we always find such q, q', a?

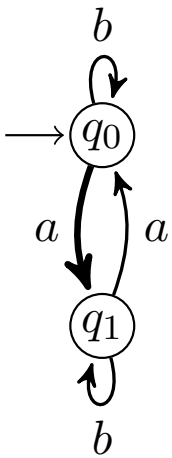
A: Not always - but then our hypothesis is wrong!

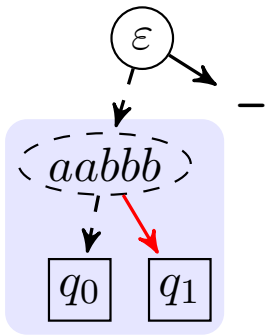
Instable Hypotheses

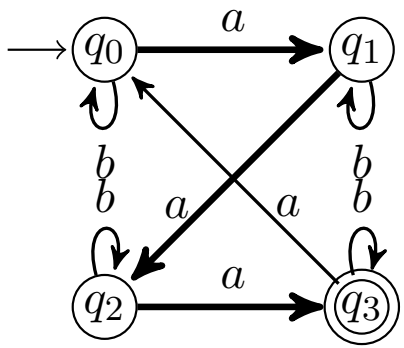


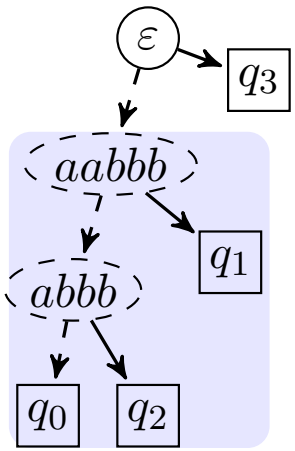
- Hypothesis contradicts correct information $\lambda(a \cdot aabbb) = 1$ (stored in DT)
- Non-canonical hypothesis
- Analyze counterexample $a \cdot aabbb/1$ ...

Instable Hypotheses



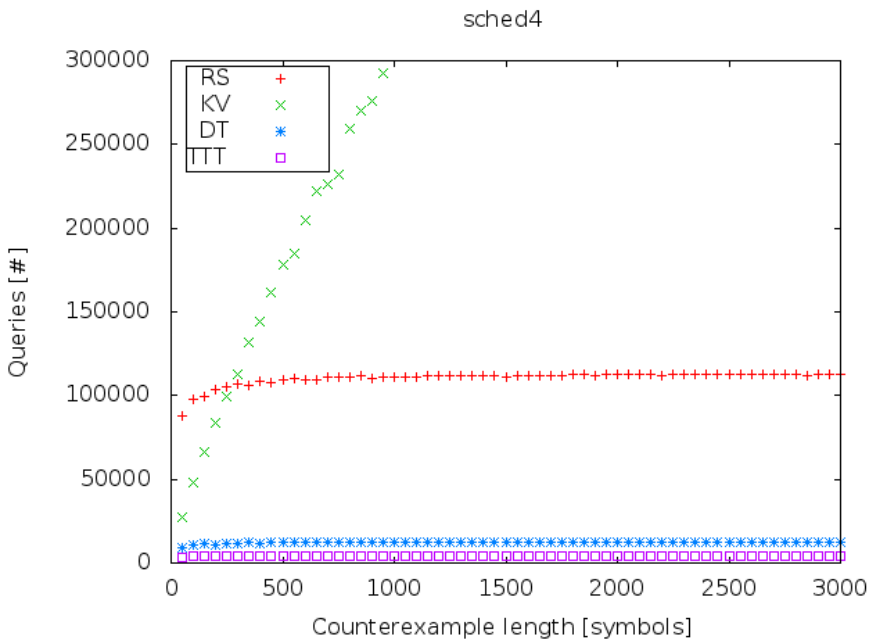


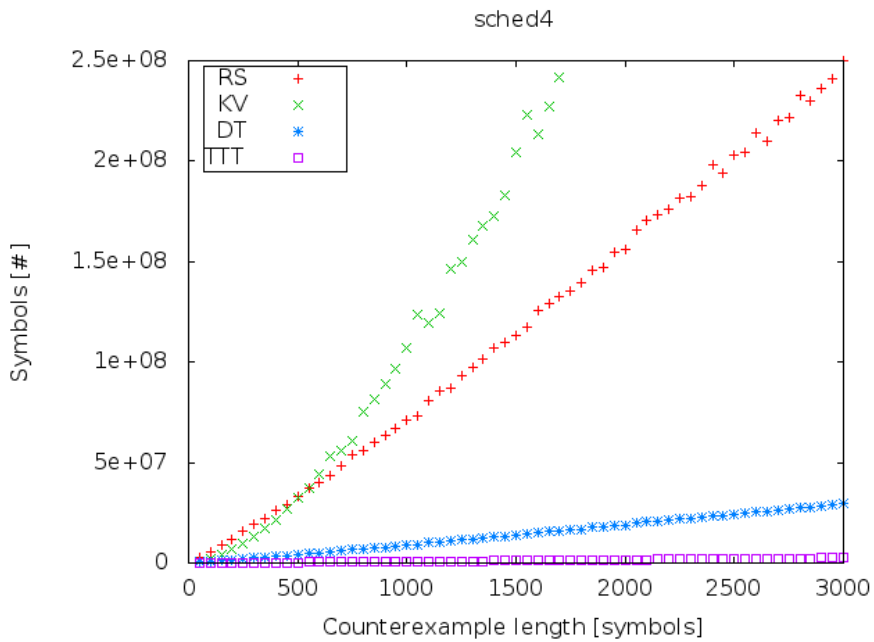




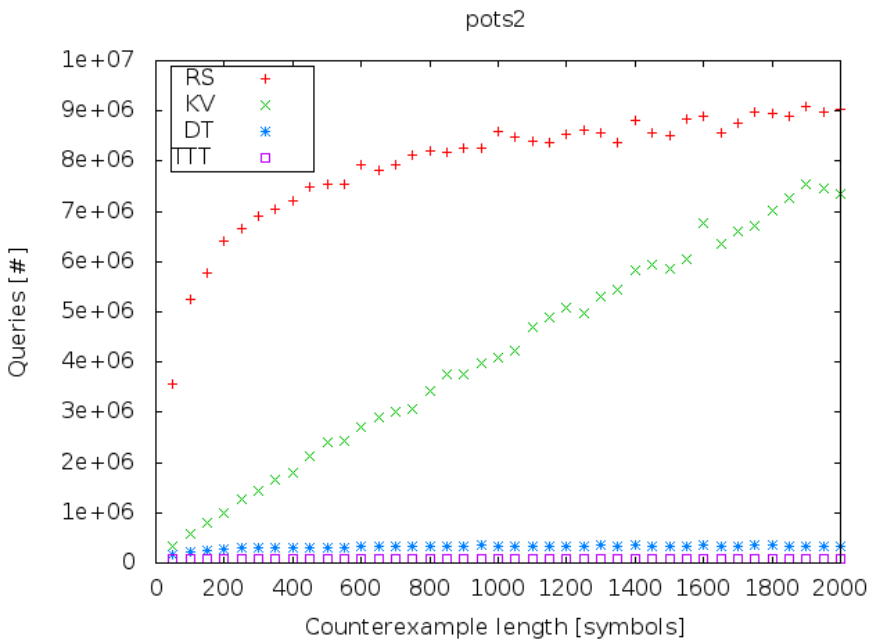
- Hypothesis contradicts correct information $\lambda(a \cdot aabbb) = 1$ (stored in DT)
- Non-canonical hypothesis
- Analyze counterexample $a \cdot aabbb/1$ ...

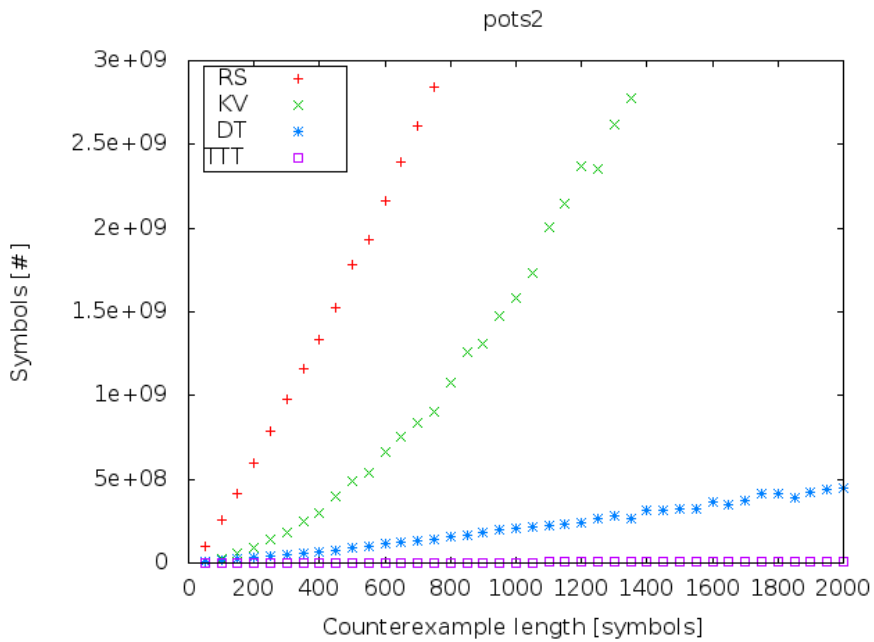
TTT – Some Results (1)





TTT – Some Results (2)





TTT – Some Results (3)

Alg.	RANDOM-10-100 ($\hat{n}/2 \leq \lvert w\rvert \leq 2\hat{n}$)			RANDOM-10-100 ($\lvert w\rvert = 500$)		
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 ($\hat{n}/2 \leq \lvert w\rvert \leq 2\hat{n}$)			RANDOM-20-200 ($\lvert w\rvert = 500$)		
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 ($\hat{n}/2 \leq \lvert w\rvert \leq 2\hat{n}$)			SCHED4 ($\lvert w\rvert = 500$)		
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

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