

Learning Algorithms and Formal Verification

P. Madhusudan

University of Illinois at Urbana-Champaign

VMCAI
Nice, 2007

A Tutorial

Outline

- 1 Computational Learning Theory
- 2 Learning Regular Languages
- 3 Applications to Verification
 - Compositional verification
 - Interface Synthesis
 - Learning Reachable Sets

Outline

- 1 Computational Learning Theory
- 2 Learning Regular Languages
- 3 Applications to Verification
 - Compositional verification
 - Interface Synthesis
 - Learning Reachable Sets

Learning Theory

- Generic theme: Learn a *concept* from a *concept class* using positive and negative instances of the concept.
- Examples:
 - Can we learn a polygon given sample points inside the polygon and outside it?
 - Can we learn a boolean function given sample evaluations?
 - Learning in presence of noise
- Applications to AI, neural networks, geometry, mining, ...
- And verification :-)

Learning Theory

- Generic theme: Learn a *concept* from a *concept class* using positive and negative instances of the concept.
- Examples:
 - Can we learn a polygon given sample points inside the polygon and outside it?
 - Can we learn a boolean function given sample evaluations?
 - Learning in presence of noise
- Applications to AI, neural networks, geometry, mining, ...
- And verification :-)

Learning Theory

- Generic theme: Learn a *concept* from a *concept class* using positive and negative instances of the concept.
- Examples:
 - Can we learn a polygon given sample points inside the polygon and outside it?
 - Can we learn a boolean function given sample evaluations?
 - Learning in presence of noise
- Applications to AI, neural networks, geometry, mining, ...
- And verification :-)

Learning Theory

- Generic theme: Learn a *concept* from a *concept class* using positive and negative instances of the concept.
- Examples:
 - Can we learn a polygon given sample points inside the polygon and outside it?
 - Can we learn a boolean function given sample evaluations?
 - Learning in presence of noise
- Applications to AI, neural networks, geometry, mining, ...
- And verification :-)

Learning Theory

- Generic theme: Learn a *concept* from a *concept class* using positive and negative instances of the concept.
- Examples:
 - Can we learn a polygon given sample points inside the polygon and outside it?
 - Can we learn a boolean function given sample evaluations?
 - Learning in presence of noise
- Applications to AI, neural networks, geometry, mining, ...
- And verification :-)

Learning Theory

- Generic theme: Learn a *concept* from a *concept class* using positive and negative instances of the concept.
- Examples:
 - Can we learn a polygon given sample points inside the polygon and outside it?
 - Can we learn a boolean function given sample evaluations?
 - Learning in presence of noise
- Applications to AI, neural networks, geometry, mining, ...
- And verification :-)

The PAC learning model

- Probably Approximately Correct Learning (Valiant'84)
 - *For any concept, ϵ, δ , we can, with probability $1 - \delta$, efficiently learn using samples an ϵ -approximation of the concept.*
 - Key concept: Vapnik-Chervonenkis (VC) dimension that captures how hard a class is to learn.
 - VC-dimension characterizes the sample size needed to learn any concept in the class.

Sample Results

- Conjunctions of boolean literals is PAC-learnable.
- 3-DNF is *not* efficiently PAC-learnable but 3-CNF is
- “If a concept class has finite VC-dimension, then it is efficiently PAC learnable.”



The PAC learning model

- Probably Approximately Correct Learning (Valiant'84)
 - *For any concept, ϵ , δ , we can, with probability $1 - \delta$, efficiently learn using samples an ϵ -approximation of the concept.*
 - Key concept: Vapnik-Chervonenkis (VC) dimension that captures how hard a class is to learn.
 - VC-dimension characterizes the sample size needed to learn any concept in the class.

Sample Results

- Conjunctions of boolean literals is PAC-learnable.
- 3-DNF is *not* efficiently PAC-learnable but 3-CNF is
- “If a concept class has finite VC-dimension, then it is efficiently PAC learnable.”



The PAC learning model

- Probably Approximately Correct Learning (Valiant'84)
 - *For any concept, ϵ , δ , we can, with probability $1 - \delta$, efficiently learn using samples an ϵ -approximation of the concept.*
 - Key concept: Vapnik-Chervonenkis (VC) dimension that captures how hard a class is to learn.
 - VC-dimension characterizes the sample size needed to learn any concept in the class.

Sample Results

- Conjunctions of boolean literals is PAC-learnable.
- 3-DNF is *not* efficiently PAC-learnable but 3-CNF is
- “If a concept class has finite VC-dimension, then it is efficiently PAC learnable.”

The PAC learning model

- Probably Approximately Correct Learning (Valiant'84)
 - *For any concept, ϵ, δ , we can, with probability $1 - \delta$, efficiently learn using samples an ϵ -approximation of the concept.*
 - Key concept: Vapnik-Chervonenkis (VC) dimension that captures how hard a class is to learn.
 - VC-dimension characterizes the sample size needed to learn any concept in the class.

Sample Results

- Conjunctions of boolean literals is PAC-learnable.
- 3-DNF is *not* efficiently PAC-learnable but 3-CNF is
- “If a concept class has finite VC-dimension, then it is efficiently PAC learnable.”

The Occam Razor

- Learning can be seen as a way of *exactly* learning a concept from glimpses of it.
- **Occam view:**
Given glimpses of a concept, what is the *simplest* description of the concept?
- **Crucial in the verification concept:**
Rather than learn a particular concept, we want to learn a *simple abstraction* of the concept.
- Learning hence gives an *algorithmic* way to abstract a concept.

The Occam Razor

- Learning can be seen as a way of *exactly* learning a concept from glimpses of it.
- **Occam view:**
Given glimpses of a concept, what is the *simplest* description of the concept?
- **Crucial in the verification concept:**
Rather than learn a particular concept, we want to learn a *simple abstraction* of the concept.
- Learning hence gives an *algorithmic* way to abstract a concept.

The Occam Razor

- Learning can be seen as a way of *exactly* learning a concept from glimpses of it.
- **Occam view:**
Given glimpses of a concept, what is the *simplest* description of the concept?
- **Crucial in the verification concept:**
Rather than learn a particular concept, we want to learn a *simple abstraction* of the concept.
- Learning hence gives an *algorithmic* way to abstract a concept.

The Occam Razor

- Learning can be seen as a way of *exactly* learning a concept from glimpses of it.
- **Occam view:**
Given glimpses of a concept, what is the *simplest* description of the concept?
- **Crucial in the verification concept:**
Rather than learn a particular concept, we want to learn a *simple abstraction* of the concept.
- Learning hence gives an *algorithmic* way to abstract a concept.

Simple concepts are at the heart of correctness

Verification of systems is a very hard problem.
But there is promise in solving it because

- The programmer isn't an adversary.
- Simple concepts usually lie behind the correctness of code.
- Examples: simple loop invariants; simple shape invariants of structures; simple predicates that control flow; simple agreements between components; simple concurrency conventions.
- These simple concepts are however *hidden*.

Learning may be a way to mine these simple concepts back,
and use them to verify the system.

Simple concepts are at the heart of correctness

Verification of systems is a very hard problem.
But there is promise in solving it because

- The programmer isn't an adversary.
- Simple concepts usually lie behind the correctness of code.
- Examples: simple loop invariants; simple shape invariants of structures; simple predicates that control flow; simple agreements between components; simple concurrency conventions.
- These simple concepts are however *hidden*.

Learning may be a way to mine these simple concepts back,
and use them to verify the system.

Simple concepts are at the heart of correctness

Verification of systems is a very hard problem.
But there is promise in solving it because

- The programmer isn't an adversary.
- Simple concepts usually lie behind the correctness of code.
- Examples: simple loop invariants; simple shape invariants of structures; simple predicates that control flow; simple agreements between components; simple concurrency conventions.
- These simple concepts are however *hidden*.

Learning may be a way to mine these simple concepts back,
and use them to verify the system.

Simple concepts are at the heart of correctness

Verification of systems is a very hard problem.
But there is promise in solving it because

- The programmer isn't an adversary.
- Simple concepts usually lie behind the correctness of code.
- Examples: simple loop invariants; simple shape invariants of structures; simple predicates that control flow; simple agreements between components; simple concurrency conventions.
- These simple concepts are however *hidden*.

Learning may be a way to mine these simple concepts back,
and use them to verify the system.

Learning and Verification

Most applications of learning in verification has concentrated on the efficient algorithms for learning *regular languages*.

- **Compositional verification:**
Learn the assumption-committment guarantees that components make within themselves.
- **Interface synthesis:**
Learn the way a programmer *intended* a piece of code to be used.
- **Reachability analysis:**
Learn a simple invariant of the set of all reachable states reached by an infinite-state system.
- **Black-box checking:**
Learn the model of a black-box system using tests, and verify it.

Learning and Verification

Most applications of learning in verification has concentrated on the efficient algorithms for learning *regular languages*.

- **Compositional verification:**
Learn the assumption-committment guarantees that components make within themselves.
- **Interface synthesis:**
Learn the way a programmer *intended* a piece of code to be used.
- **Reachability analysis:**
Learn a simple invariant of the set of all reachable states reached by an infinite-state system.
- **Black-box checking:**
Learn the model of a black-box system using tests, and verify it.

Structure of the talk

- Angluin's algorithm for learning regular languages efficiently
- Compositional verification using learning
- Other applications in verification
- Future directions with the Occam razor

Structure of the talk

- Angluin's algorithm for learning regular languages efficiently
- Compositional verification using learning
- Other applications in verification
- Future directions with the Occam razor

Outline

- 1 Computational Learning Theory
- 2 Learning Regular Languages
- 3 Applications to Verification
 - Compositional verification
 - Interface Synthesis
 - Learning Reachable Sets

Learning Regular Languages

Fix a finite alphabet Σ .

- There is a learner and a teacher
- Teacher knows a regular language T
- **Objective of the learner** : To learn T by constructing an automaton for T .

Complexity will be measured on the complexity of the language: the minimum number of states needed to capture T .

Learning Regular Languages

Fix a finite alphabet Σ .

- There is a learner and a teacher
- Teacher knows a regular language T
- **Objective of the learner** : To learn T by constructing an automaton for T .

Complexity will be measured on the complexity of the language: the minimum number of states needed to capture T .

Learning Regular Languages

Different learning models:

Passive learning Given a set of positive and negative examples, learn the language.

[Gold]: NP-complete.

PAC learning Probably approximately correct (PAC)
Regular languages are not efficiently PAC-learnable
But are with membership queries.

Active learning [Angluin'86] Learner allowed to *ask* questions:

- Membership questions: Is $w \in T$?
Answer: Yes/No
- Equivalence question: Is $T = L(C)$?
Answer: Yes, or (No, counterexample)
Counterexample is in $(T \setminus L(C)) \cup (L(C) \setminus T)$.

Active Learning

Theorem (Angluin, Rivest-Schapire, Kearns-Vazirani)

Regular languages can be learnt using at most $O(kn^2 + n \log m)$ membership and $O(n)$ equivalence queries.

- *n is size of the minimal DFA accepting target language T*
- *m is the size of the largest counterexample*
- *k is the size of the alphabet.*

Also, in time polynomial in $O(kn^2 + n \log m)$.

So, how do we learn T ?

Key points:

- How many states are there?
- How do we reach these states from the initial state?
- How do we build the transitions correctly?

When are states different?

Simple observation:

Let u and v be two strings.

If $\exists w$ such that $uw \in T \iff vw \notin T$,

then

u and v must lead to *different* states.

Let's say u and v are *distinguishable* if the above condition holds.

To know that automaton for T has n states, we will find n strings s_1, \dots, s_n , that are pairwise distinguishable.

When are states different?

Simple observation:

Let u and v be two strings.

If $\exists w$ such that $uw \in T \iff vw \notin T$,

then

u and v must lead to *different* states.

Let's say u and v are *distinguishable* if the above condition holds.

To know that automaton for T has n states, we will find n strings s_1, \dots, s_n , that are pairwise distinguishable.

When are states different?

Simple observation:

Let u and v be two strings.

If $\exists w$ such that $uw \in T \iff vw \notin T$,

then

u and v must lead to *different* states.

Let's say u and v are *distinguishable* if the above condition holds.

To know that automaton for T has n states, we will find n strings s_1, \dots, s_n , that are pairwise distinguishable.

Access strings

Access string to a state q :

Some string that gets you from q_0 to q .

Hence ϵ is an access string for q_0 .

So, if we have n access strings s_1, \dots, s_n , that are pairwise distinguishable, then the states reached on these strings must *all* be different.

Access strings

Access string to a state q :

Some string that gets you from q_0 to q .

Hence ϵ is an access string for q_0 .

So, if we have n access strings s_1, \dots, s_n , that are pairwise distinguishable, then the states reached on these strings must *all* be different.

An observation pack

Access strings	s_1	s_2	s_k
Experiments	E_{s_1}	E_{s_2}	E_{s_k}

An *observation pack* for T has n access strings $S = \{s_1, \dots, s_n\}$, and each $s \in S$ is associated with a set of experiments E_s such that:

- Each E_{s_i} consists of a set of pairs of the form $(u, +)$ or $(u, -)$:
 - $(u, +) \in E_{s_i}$ implies $s_i \cdot u_i \in T$
 - $(u, -) \in E_{s_i}$ implies $s_i \cdot u_i \notin T$
- For any two access strings s_i and s_j , there is some experiment that distinguishes them.
 i.e. there is some u that figures in E_{s_i} and E_{s_j} with opposite polarity.
- $\epsilon \in S$, and $\epsilon \in E_{s_i}$ for each i .

Note: If an observation pack with n access strings exists, then minimal automaton for T has *at least* n states.

An observation pack

Access strings	s_1	s_2	s_k
Experiments	E_{s_1}	E_{s_2}	E_{s_k}

An *observation pack* for T has n access strings $S = \{s_1, \dots, s_n\}$, and each $s \in S$ is associated with a set of experiments E_s such that:

- Each E_{s_i} consists of a set of pairs of the form $(u, +)$ or $(u, -)$:
 - $(u, +) \in E_{s_i}$ implies $s_i \cdot u_i \in T$
 - $(u, -) \in E_{s_i}$ implies $s_i \cdot u_i \notin T$
- For any two access strings s_i and s_j , there is some experiment that distinguishes them.
 i.e. there is some u that figures in E_{s_i} and E_{s_j} with opposite polarity.
- $\epsilon \in S$, and $\epsilon \in E_{s_i}$ for each i .

Note: If an observation pack with n access strings exists, then minimal automaton for T has *at least* n states.

An observation pack

Access strings	s_1	s_2	s_k
Experiments	E_{s_1}	E_{s_2}	E_{s_k}

An *observation pack* for T has n access strings $S = \{s_1, \dots, s_n\}$, and each $s \in S$ is associated with a set of experiments E_s such that:

- Each E_{s_i} consists of a set of pairs of the form $(u, +)$ or $(u, -)$:
 - $(u, +) \in E_{s_i}$ implies $s_i \cdot u_i \in T$
 - $(u, -) \in E_{s_i}$ implies $s_i \cdot u_i \notin T$
- For any two access strings s_i and s_j , there is some experiment that distinguishes them.
 i.e. there is some u that figures in E_{s_i} and E_{s_j} with opposite polarity.
- $\epsilon \in S$, and $\epsilon \in E_{s_i}$ for each i .

Note: If an observation pack with n access strings exists, then minimal automaton for T has *at least* n states.

An observation pack

Access strings	s_1	s_2	s_k
Experiments	E_{s_1}	E_{s_2}	E_{s_k}

An *observation pack* for T has n access strings $S = \{s_1, \dots, s_n\}$, and each $s \in S$ is associated with a set of experiments E_s such that:

- Each E_{s_i} consists of a set of pairs of the form $(u, +)$ or $(u, -)$:
 - $(u, +) \in E_{s_i}$ implies $s_i \cdot u_i \in T$
 - $(u, -) \in E_{s_i}$ implies $s_i \cdot u_i \notin T$
- For any two access strings s_i and s_j , there is some experiment that distinguishes them.
 i.e. there is some u that figures in E_{s_i} and E_{s_j} with opposite polarity.
- $\epsilon \in S$, and $\epsilon \in E_{s_i}$ for each i .

Note: If an observation pack with n access strings exists, then minimal automaton for T has *at least* n states.

An observation pack

Access strings	s_1	s_2	s_k
Experiments	E_{s_1}	E_{s_2}	E_{s_k}

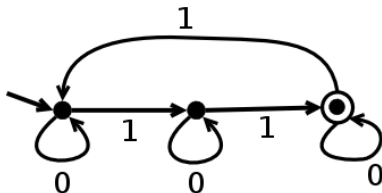
An *observation pack* for T has n access strings $S = \{s_1, \dots, s_n\}$, and each $s \in S$ is associated with a set of experiments E_s such that:

- Each E_{s_i} consists of a set of pairs of the form $(u, +)$ or $(u, -)$:
 - $(u, +) \in E_{s_i}$ implies $s_i \cdot u_i \in T$
 - $(u, -) \in E_{s_i}$ implies $s_i \cdot u_i \notin T$
- For any two access strings s_i and s_j , there is some experiment that distinguishes them.
 i.e. there is some u that figures in E_{s_i} and E_{s_j} with opposite polarity.
- $\epsilon \in S$, and $\epsilon \in E_{s_i}$ for each i .

Note: If an observation pack with n access strings exists, then minimal automaton for T has *at least* n states.

Example

Target language T : strings over $\{0, 1\}$ where $\#1\text{'s} = 2 \pmod 3$



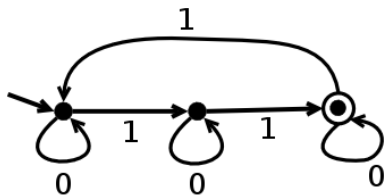
An observation pack:

Access strings	ϵ	010
Experiments	$(\epsilon, -)$ $(10, -)$	$(\epsilon, -)$ $(10, +)$

$\epsilon.\epsilon \notin T$; $010.\epsilon \notin T$
 $\epsilon.10 \notin T$; $010.10 \in T$

Example

Target language T : strings over $\{0, 1\}$ where $\#1's = 2 \pmod 3$



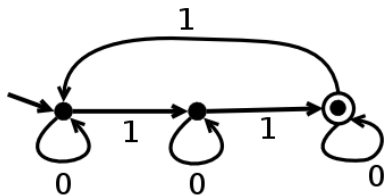
An observation pack:

Access strings	ϵ	010
Experiments	$(\epsilon, -)$ $(10, -)$	$(\epsilon, -)$ $(10, +)$

$\epsilon.\epsilon \notin T$; $010.\epsilon \notin T$
 $\epsilon.10 \notin T$; $010.10 \in T$

Example

Target language T : strings over $\{0, 1\}$ where $\#1\text{'s} = 2 \pmod 3$



An observation pack:

Access strings	ϵ	010
Experiments	$(\epsilon, -)$ $(10, -)$	$(\epsilon, -)$ $(10, +)$

$\epsilon.\epsilon \notin T$; $010.\epsilon \notin T$
 $\epsilon.10 \notin T$; $010.10 \in T$

Like-ness and escape

Let O be an observation pack.

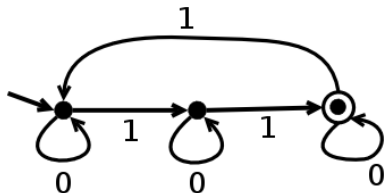
A word w is *like* an access string s in O , if w agrees with s on all the experiments in E_s .

I.e. $\forall u \in E_s, wu \in T$ iff $su \in T$.

Note: w can be like *at most one* access string in O .
(since no two access strings are alike).

If w is *not* like any access string, we say it *escapes* the pack.

Example



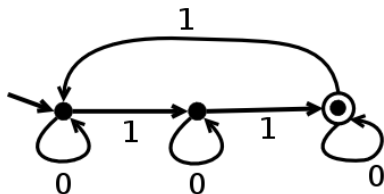
Access strings	ϵ	110
Experiments	$(\epsilon, -)$ $(10, -)$	$(\epsilon, -)$ $(10, +)$

The word 001 is like 010 (since $001.\epsilon \notin T$, $001.10 \in T$).

The word 11 is not like any access string in O (since $11.\epsilon \in T$).

So 11 *escapes*.

Example



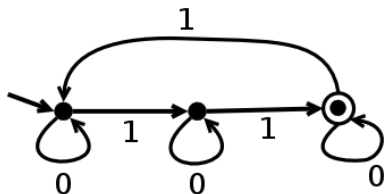
Access strings	ϵ	110
Experiments	$(\epsilon, -)$ $(10, -)$	$(\epsilon, -)$ $(10, +)$

The word 001 is like 010 (since $001.\epsilon \notin T$, $001.10 \in T$).

The word 11 is not like any access string in O (since $11.\epsilon \in T$).

So 11 *escapes*.

Example



Access strings	ϵ	110
Experiments	$(\epsilon, -)$ $(10, -)$	$(\epsilon, -)$ $(10, +)$

The word 001 is like 010 (since $001.\epsilon \notin T$, $001.10 \in T$).

The word 11 is not like any access string in O (since $11.\epsilon \in T$).

So 11 *escapes*.

Expanding a pack

If O is an observation pack, and w escapes O , then we can expand O to include w :

- Add w as a new access string
- For every access s string in O , there is some u in E_s that distinguishes w and s .
- Add this string to E_w

The new pack is a proper observation pack.
And has one more access string.

Closure

An observation pack O is said to be *closed* if

- For every access string s in O and $a \in \Sigma$, $s.a$ is like some access string in O .

If O is closed, we can build an automaton from it:

- States: The access strings in O : $\{s_1 \dots, s_k\}$
- From s on a , go to the state that is *like* sa .
- Mark a state s final iff $(\epsilon, +) \in E_s$.

Closure

An observation pack O is said to be *closed* if

- For every access string s in O and $a \in \Sigma$, $s.a$ is like some access string in O .

If O is closed, we can build an automaton from it:

- States: The access strings in O : $\{s_1 \dots, s_k\}$
- From s on a , go to the state that is *like* sa .
- Mark a state s final iff $(\epsilon, +) \in E_s$.

Closure

An observation pack O is said to be *closed* if

- For every access string s in O and $a \in \Sigma$, $s.a$ is like some access string in O .

If O is closed, we can build an automaton from it:

- States: The access strings in O : $\{s_1 \dots, s_k\}$
- From s on a , go to the state that is *like* sa .
- Mark a state s final iff $(\epsilon, +) \in E_s$.

Closure

An observation pack O is said to be *closed* if

- For every access string s in O and $a \in \Sigma$, $s.a$ is like some access string in O .

If O is closed, we can build an automaton from it:

- States: The access strings in O : $\{s_1 \dots, s_k\}$
- From s on a , go to the state that is *like* sa .
- Mark a state s final iff $(\epsilon, +) \in E_s$.

Automaton construction

Theorem

If the observation pack O has as many states as M_T , then the automaton constructed is isomorphic to M_T .

Proof.

- The number of states is correct.
- Initial state maps to initial state of M_T .
- On any letter, we move to the right state.
- Final states are marked correctly.



So, the whole problem reduces to finding an observation pack with n access strings!!

Automaton construction

Theorem

If the observation pack O has as many states as M_T , then the automaton constructed is isomorphic to M_T .

Proof.

- The number of states is correct.
- Initial state maps to initial state of M_T .
- On any letter, we move to the right state.
- Final states are marked correctly.



So, the whole problem reduces to finding an observation pack with n access strings!!

Automaton construction

Theorem

If the observation pack O has as many states as M_T , then the automaton constructed is isomorphic to M_T .

Proof.

- The number of states is correct.
- Initial state maps to initial state of M_T .
- On any letter, we move to the right state.
- Final states are marked correctly.



So, the whole problem reduces to finding an observation pack with n access strings!!

Learning from a false automaton

Let O be an observation pack.

Phase I: If O is *not* closed, expand pack using some new access string $s.a$.

Phase II: If O is closed, and has less access strings than $|M_T|$.

Then automaton constructed has too few states.
How do we learn access strings to new states?

Equivalence query:

- Build conjecture automaton C .
- Ask teacher whether “ $L(C) = T$?”
- Use counterexample given by teacher to generate new access string.

Learning from a false automaton

Let O be an observation pack.

Phase I: If O is *not* closed, expand pack using some new access string $s.a$.

Phase II: If O is closed, and has less access strings than $|M_T|$.

Then automaton constructed has too few states.
How do we learn access strings to new states?

Equivalence query:

- Build conjecture automaton C .
- Ask teacher whether " $L(C) = T$?"
- Use counterexample given by teacher to generate new access string.

Learning from a false automaton

Let O be an observation pack.

Phase I: If O is *not* closed, expand pack using some new access string $s.a$.

Phase II: If O is closed, and has less access strings than $|M_T|$.

Then automaton constructed has too few states.
How do we learn access strings to new states?

Equivalence query:

- Build conjecture automaton C .
- Ask teacher whether " $L(C) = T$?"
- Use counterexample given by teacher to generate new access string.

Learning from a false automaton

Let O be an observation pack.

Phase I: If O is *not* closed, expand pack using some new access string $s.a$.

Phase II: If O is closed, and has less access strings than $|M_T|$.

Then automaton constructed has too few states.
How do we learn access strings to new states?

Equivalence query:

- Build conjecture automaton C .
- Ask teacher whether " $L(C) = T$?"
- Use counterexample given by teacher to generate new access string.

Learning from a false automaton

Let O be an observation pack.

Phase I: If O is *not* closed, expand pack using some new access string $s.a$.

Phase II: If O is closed, and has less access strings than $|M_T|$.

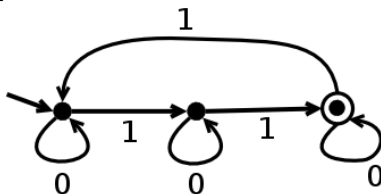
Then automaton constructed has too few states.
How do we learn access strings to new states?

Equivalence query:

- Build conjecture automaton C .
- Ask teacher whether “ $L(C) = T$?”
- Use counterexample given by teacher to generate new access string.

A learning example

Target language T



Access strings	$s_0 = \epsilon$
Experiments	$(\epsilon, -)$

Check closure:

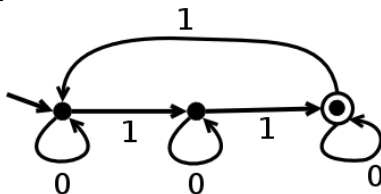
0 is like ϵ (since $0 \notin T$).

1 is like ϵ (since $1 \notin T$).



A learning example

Target language T



Access strings	$s_0 = \epsilon$
Experiments	$(\epsilon, -)$

Check closure:

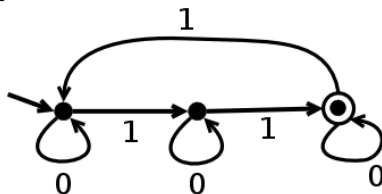
0 is like ϵ (since $0 \notin T$).

1 is like ϵ (since $1 \notin T$).



A learning example

Target language T



Access strings	$s_0 = \epsilon$
Experiments	$(\epsilon, -)$

Check closure:

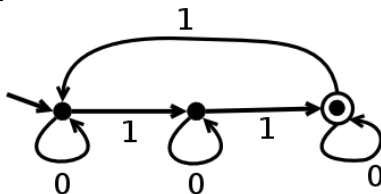
0 is like ϵ (since $0 \notin T$).

1 is like ϵ (since $1 \notin T$).



A learning example

Target language T



Access strings	$s_0 = \epsilon$
Experiments	$(\epsilon, -)$

Check closure:

0 is like ϵ (since $0 \notin T$).

1 is like ϵ (since $1 \notin T$).



A learning example



Counter-example: $101 \in T \setminus C$

Run of 101 on C:

$$s_0 \xrightarrow{1} s_0 \xrightarrow{0} s_0 \xrightarrow{1} s_0$$

- $s_0 = \epsilon$
- $s_0.101 \in T$
- $s_0.01 \notin T$.
- So we cannot go on 1 to s_0 !
(since 01 distinguishes 1 and s_0)
- So let's add 01 as experiment string for s_0 .

A learning example



Counter-example: $101 \in T \setminus C$

Run of 101 on C:

$$s_0 \xrightarrow{1} s_0 \xrightarrow{0} s_0 \xrightarrow{1} s_0$$

- $s_0 = \epsilon$
- $s_0.101 \in T$
- $s_0.01 \notin T$.
- So we cannot go on 1 to s_0 !
(since 01 distinguishes 1 and s_0)
- So let's add 01 as experiment string for s_0 .

A learning example



Counter-example: $101 \in T \setminus C$

Run of 101 on C:

$$s_0 \xrightarrow{1} s_0 \xrightarrow{0} s_0 \xrightarrow{1} s_0$$

- $s_0 = \epsilon$
- $s_0.101 \in T$
- $s_0.01 \notin T$.
- So we cannot go on 1 to s_0 !
(since 01 distinguishes 1 and s_0)
- So let's add 01 as experiment string for s_0 .

A learning example



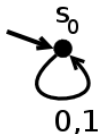
Counter-example: $101 \in T \setminus C$

Run of 101 on C:

$$s_0 \xrightarrow{1} s_0 \xrightarrow{0} s_0 \xrightarrow{1} s_0$$

- $s_0 = \epsilon$
- $s_0.101 \in T$
- $s_0.01 \notin T$.
- So we cannot go on 1 to s_0 !
(since 01 distinguishes 1 and s_0)
- So let's add 01 as experiment string for s_0 .

A learning example



Counter-example: $101 \in T \setminus C$

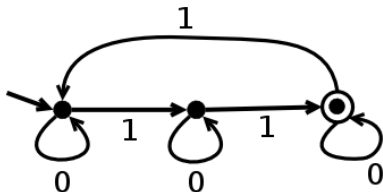
Run of 101 on C :

$s_0 \xrightarrow{1} s_0 \xrightarrow{0} s_0 \xrightarrow{1} s_0$

- $s_0 = \epsilon$
- $s_0.101 \in T$
- $s_0.01 \notin T$.
- So we cannot go on 1 to s_0 !
(since 01 distinguishes 1 and s_0)
- So let's add 01 as experiment string for s_0 .

A learning example

T:



Access strings	$s_0 = \epsilon$
Experiments	$(\epsilon, -)$ $(01, -)$

Check closure:

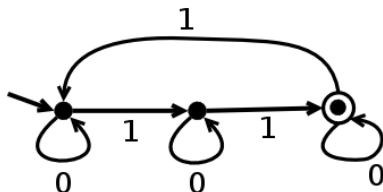
0 is like ϵ (since $0 \notin T$; $010 \notin T$).

But 1 is *not* like ϵ (since $1.01 \in T$).

So 1 escapes; and becomes a new access string.

A learning example

T:



Access strings	$S_0 = \epsilon$
Experiments	$(\epsilon, -)$ $(01, -)$

Check closure:

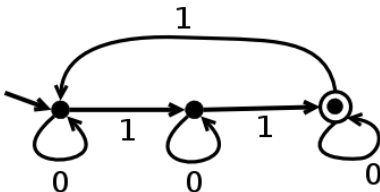
0 is like ϵ (since $0 \notin T$; $010 \notin T$).

But 1 is *not* like ϵ (since $1.01 \in T$).

So 1 escapes; and becomes a new access string

A learning example

T:



Access strings	$s_0 = \epsilon$
Experiments	$(\epsilon, -)$ $(01, -)$

Check closure:

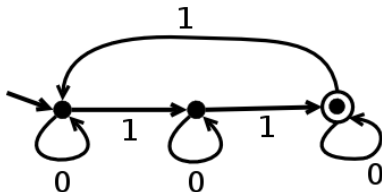
0 is like ϵ (since $0 \notin T$; $010 \notin T$).

But 1 is *not* like ϵ (since $1.01 \in T$).

So 1 escapes; and becomes a new access string.

A learning example

T:



Access strings	$s_0 = \epsilon$
Experiments	$(\epsilon, -)$ $(01, -)$

Check closure:

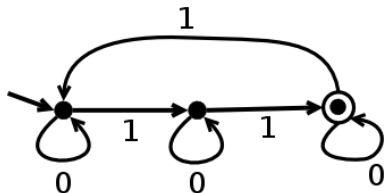
0 is like ϵ (since $0 \notin T$; $010 \notin T$).

But 1 is *not* like ϵ (since $1.01 \in T$).

So 1 escapes; and becomes a new access string.

A learning example

T:



Access strings	$s_0 = \epsilon$	$s_1 = 1$
Experiments	$(\epsilon, -)$	$(\epsilon, -)$
	$(01, -)$	$(01, +)$

Check closure:

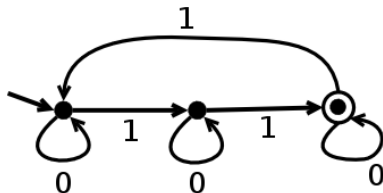
10 is like 1 (since $10 \notin T$ and $10.01 \in T$)

But 11 is neither like ϵ nor like 1 (since $11 \in T$).

So 11 escapes and forms a new access string.

A learning example

T:



Access strings	$s_0 = \epsilon$	$s_1 = 1$
Experiments	$(\epsilon, -)$	$(\epsilon, -)$
	$(01, -)$	$(01, +)$

Check closure:

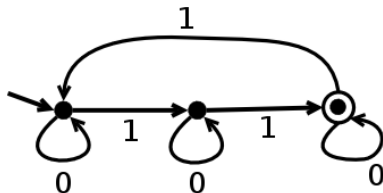
10 is like 1 (since $10 \notin T$ and $10.01 \in T$)

But 11 is neither like ϵ nor like 1 (since $11 \in T$).

So 11 escapes and forms a new access string.

A learning example

T:



Access strings	$s_0 = \epsilon$	$s_1 = 1$
Experiments	$(\epsilon, -)$	$(\epsilon, -)$
	$(01, -)$	$(01, +)$

Check closure:

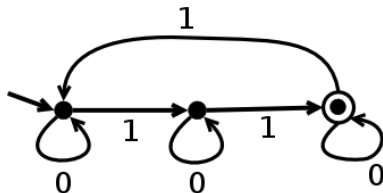
10 is like 1 (since $10 \notin T$ and $10.01 \in T$)

But 11 is neither like ϵ nor like 1 (since $11 \in T$).

So 11 escapes and forms a new access string.

A learning example

T:



Access strings	$s_0 = \epsilon$	$s_1 = 1$
Experiments	$(\epsilon, -)$	$(\epsilon, -)$
	$(01, -)$	$(01, +)$

Check closure:

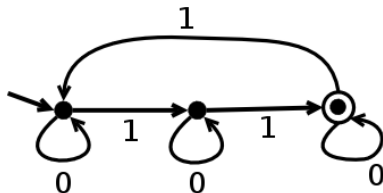
10 is like 1 (since $10 \notin T$ and $10.01 \in T$)

But 11 is neither like ϵ nor like 1 (since $11 \in T$).

So 11 escapes and forms a new access string.

A learning example

T:



Access strings	$s_0 = \epsilon$	$s_1 = 1$
Experiments	$(\epsilon, -)$	$(\epsilon, -)$
	$(01, -)$	$(01, +)$

Check closure:

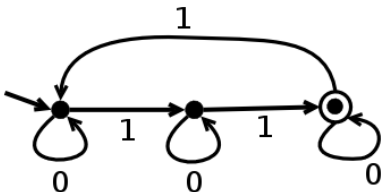
10 is like 1 (since $10 \notin T$ and $10.01 \in T$)

But 11 is neither like ϵ nor like 1 (since $11 \in T$).

So 11 escapes and forms a new access string.

A learning example

T:



Access strings	$s_0 = \epsilon$	$s_1 = 1$	$s_2 = 11$
Experiments	$(\epsilon, -)$ $(01, -)$	$(\epsilon, -)$ $(01, +)$	$(\epsilon, +)$

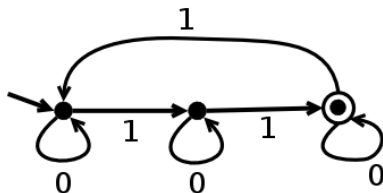
Check closure:

0 is like s_0 ; 10 is like 1;

110 is like 11; 111 is like 0.

A learning example

T:



Access strings	$s_0 = \epsilon$	$s_1 = 1$	$s_2 = 11$
Experiments	$(\epsilon, -)$ $(01, -)$	$(\epsilon, -)$ $(01, +)$	$(\epsilon, +)$

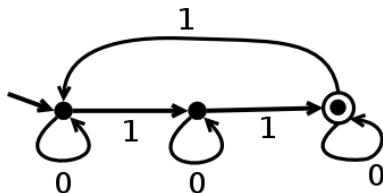
Check closure:

0 is like s_0 ; 10 is like 1;

110 is like 11; 111 is like 0.

A learning example

T:



Access strings	$s_0 = \epsilon$	$s_1 = 1$	$s_2 = 11$
Experiments	$(\epsilon, -)$ $(01, -)$	$(\epsilon, -)$ $(01, +)$	$(\epsilon, +)$

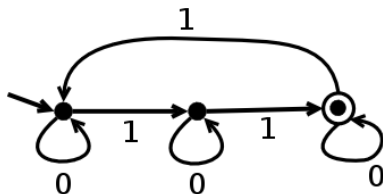
Check closure:

0 is like s_0 ; 10 is like 1;

110 is like 11; 111 is like 0.

A learning example

T:

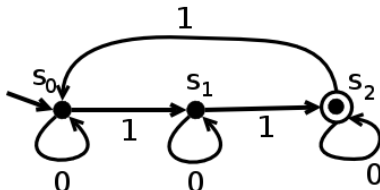


Access strings	$s_0 = \epsilon$	$s_1 = 1$	$s_2 = 11$
Experiments	$(\epsilon, -)$ $(01, -)$	$(\epsilon, -)$ $(01, +)$	$(\epsilon, +)$

Check closure:

0 is like s_0 ; 10 is like 1;

110 is like 11; 111 is like 0







Learning algorithms

- Angluin '86
 - Observation pack: table
 - Uniform experiment sets
 - Access strings are prefix closed.
 - Each time the conjecture machine *avoids* the earlier counterexamples.
 - Number of queries: $O(kn^2m)$
- Rivest-Schapire '93 and Kearns-Vazirani '94
 - Observation pack: RS: reduced table; KV: discrimination tree
 - Conjecture machine *may not avoid* earlier counterexamples!
 - Number of queries: $O(kn^2 + nlogm)$

Learning algorithms

- Angluin '86
 - Observation pack: table
 - Uniform experiment sets
 - Access strings are prefix closed.
 - Each time the conjecture machine *avoids* the earlier counterexamples.
 - Number of queries: $O(kn^2m)$
- Rivest-Schapire '93 and Kearns-Vazirani '94
 - Observation pack: RS: reduced table; KV: discrimination tree
 - Conjecture machine *may not avoid* earlier counterexamples!
 - Number of queries: $O(kn^2 + n \log m)$

References

-  [Angluin](#) Learning regular sets from queries and counterexamples ; [Inf. and Comp. '87](#) .
-  [Rivest, Schapire](#) Inference of finite automata using homing sequences ; [Inf. and Comp. '95](#) .
-  [Kearns, Vazirani](#): Introduction to Computational Learning Theory ; [MIT Press](#) .
-  [Balcázar, Díaz, Gavalda, Watanabe](#) Algorithms for Learning Finite Automata from queries: A Unified View ; [Tech report](#) .

Outline

- 1 Computational Learning Theory
- 2 Learning Regular Languages
- 3 Applications to Verification**
 - Compositional verification
 - Interface Synthesis
 - Learning Reachable Sets

Applications of learning reg lang to verification

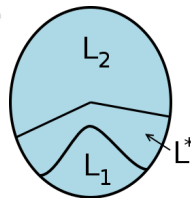
- Compositional verification (assume-guarantee reasoning):
Learning from a range
- Interface synthesis:
Solving games
- Learning reachable sets in infinite state verification
- Black-box checking
Learning and conformance testing

Learning from a range

Problem

Given two languages L_1 and L_2 , $L_1 \subseteq L_2$, can we learn a language L^* such that $L_1 \subseteq L^* \subseteq L_2$?

- Ideally, we want to learn the *smallest language* L^* lying between L_1 and L_2 .
- L_1 and L_2 can be LARGE while L^* is small:
 - L_1 – all strings with prime # of a 's
 - L_2 – all strings with odd # of a 's or prime number of b 's
 - L^* – all strings with odd number of a 's.
- Assume L_1 and L_2 are regular and teacher knows them and can answer questions.
- How do we learn L^* ?

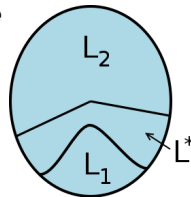


Learning from a range

Problem

Given two languages L_1 and L_2 , $L_1 \subseteq L_2$, can we learn a language L^* such that $L_1 \subseteq L^* \subseteq L_2$?

- Ideally, we want to learn the *smallest language* L^* lying between L_1 and L_2 .
- L_1 and L_2 can be LARGE while L^* is small:
 - L_1 – all strings with prime # of a 's
 - L_2 – all strings with odd # of a 's or prime number of b 's
 - L^* – all strings with odd number of a 's.
- Assume L_1 and L_2 are regular and teacher knows them and can answer questions.
- How do we learn L^* ?

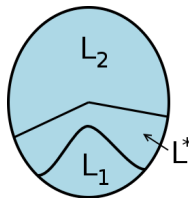


Learning from a range

Problem

Given two languages L_1 and L_2 , $L_1 \subseteq L_2$, can we learn a language L^* such that $L_1 \subseteq L^* \subseteq L_2$?

- Ideally, we want to learn the *smallest language* L^* lying between L_1 and L_2 .
- L_1 and L_2 can be LARGE while L^* is small:
 - L_1 – all strings with prime # of a 's
 - L_2 – all strings with odd # of a 's or prime number of b 's
 - L^* – all strings with odd number of a 's.
- Assume L_1 and L_2 are regular and teacher knows them and can answer questions.
- How do we learn L^* ?

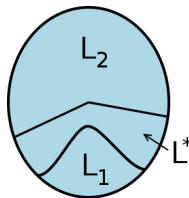


Learning from a range

Problem

Given two languages L_1 and L_2 , $L_1 \subseteq L_2$, can we learn a language L^* such that $L_1 \subseteq L^* \subseteq L_2$?

- Ideally, we want to learn the *smallest language* L^* lying between L_1 and L_2 .
- L_1 and L_2 can be LARGE while L^* is small:
 - L_1 – all strings with prime # of a 's
 - L_2 – all strings with odd # of a 's or prime number of b 's
 - L^* – all strings with odd number of a 's.
- Assume L_1 and L_2 are regular and teacher knows them and can answer questions.
- How do we learn L^* ?

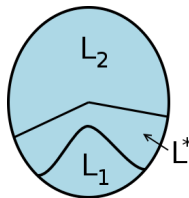


Learning from a range

Problem

Given two languages L_1 and L_2 , $L_1 \subseteq L_2$, can we learn a language L^* such that $L_1 \subseteq L^* \subseteq L_2$?

- Ideally, we want to learn the *smallest language* L^* lying between L_1 and L_2 .
- L_1 and L_2 can be LARGE while L^* is small:
 - L_1 – all strings with prime # of a 's
 - L_2 – all strings with odd # of a 's or prime number of b 's
 - L^* – all strings with odd number of a 's.
- Assume L_1 and L_2 are regular and teacher knows them and can answer questions.
- How do we learn L^* ?



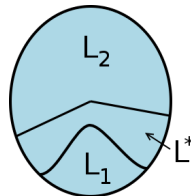
Learning from a range

Intractable

Learning the minimal L^* is NP-complete!

A Heuristic approach

- Teacher *pretends* to know L^* and asks learner to learn it using standard learning
- How does the teacher answer queries?
- Equivalence queries are easy:
 - Check if $L_1 \subseteq L(C) \subseteq L_2$.
 - If it is then done! Else return CE.
- Membership queries are hard:
 - If $w \in L_1$, then say YES!
 - If $w \notin L_2$, then say NO!
 - If $w \in L_2$ but $w \notin L_1$ — then ???



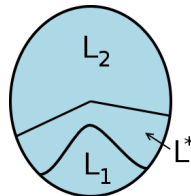
Learning from a range

Intractable

Learning the minimal L^* is NP-complete!

A Heuristic approach

- Teacher *pretends* to know L^* and asks learner to learn it using standard learning
- How does the teacher answer queries?
- Equivalence queries are easy:
 - Check if $L_1 \subseteq L(C) \subseteq L_2$.
 - If it is then done! Else return CE.
- Membership queries are hard:
 - If $w \in L_1$, then say YES!
 - If $w \notin L_2$, then say NO!
 - If $w \in L_2$ but $w \notin L_1$ — then ???



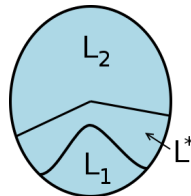
Learning from a range

Intractable

Learning the minimal L^* is NP-complete!

A Heuristic approach

- Teacher *pretends* to know L^* and asks learner to learn it using standard learning
- How does the teacher answer queries?
- Equivalence queries are easy:
 - Check if $L_1 \subseteq L(C) \subseteq L_2$.
 - If it is then done! Else return CE.
- Membership queries are hard:
 - If $w \in L_1$, then say YES!
 - If $w \notin L_2$, then say NO!
 - If $w \in L_2$ but $w \notin L_1$ — then ???



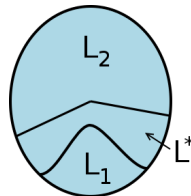
Learning from a range

Intractable

Learning the minimal L^* is NP-complete!

A Heuristic approach

- Teacher *pretends* to know L^* and asks learner to learn it using standard learning
- How does the teacher answer queries?
- Equivalence queries are easy:
 - Check if $L_1 \subseteq L(C) \subseteq L_2$.
 - If it is then done! Else return CE.
- Membership queries are hard:
 - If $w \in L_1$, then say YES!
 - If $w \notin L_2$, then say NO!
 - If $w \in L_2$ but $w \notin L_1$ — then ???



Learning from a range

Heuristic: Answer the ambiguous membership query with respect to L_1

- Results in an algorithm that does not find the minimal L^*
- But works very well in practice in finding small L^* 's.
- Worst-case: L^* may be as large as L_1 .

Learning from a range

Heuristic: Answer the ambiguous membership query with respect to L_1

- Results in an algorithm that does not find the minimal L^*
- But works very well in practice in finding small L^* 's.
- Worst-case: L^* may be as large as L_1 .

Compositional verification

Assume finite-state systems A and B .

To check if $A \parallel B \models \varphi$.

- Assume for now that φ is a property only about A .
- A may not satisfy φ in a general environment.
But satisfy φ *assuming* its environment behaves in a particular way.
- B may meet these assumptions.

Assume guarantee reasoning

Find some C such that:

- $A \parallel C \models \varphi$
- $B \preceq C$

Key observation: C could be *much smaller* than A and B !

Compositional verification

Assume finite-state systems A and B .

To check if $A \parallel B \models \varphi$.

- Assume for now that φ is a property only about A .
- A may not satisfy φ in a general environment.
But satisfy φ *assuming* its environment behaves in a particular way.
- B may meet these assumptions.

Assume guarantee reasoning

Find some C such that:

- $A \parallel C \models \varphi$
- $B \preceq C$

Key observation: C could be *much smaller* than A and B !

Compositional verification

Assume finite-state systems A and B .

To check if $A \parallel B \models \varphi$.

- Assume for now that φ is a property only about A .
- A may not satisfy φ in a general environment.
 But satisfy φ *assuming* its environment behaves in a particular way.
- B may meet these assumptions.

Assume guarantee reasoning

Find some C such that:

- $A \parallel C \models \varphi$
- $B \preceq C$

Key observation: C could be *much smaller* than A and B !

Compositional verification

Assume finite-state systems A and B .

To check if $A \parallel B \models \varphi$.

- Assume for now that φ is a property only about A .
- A may not satisfy φ in a general environment.
But satisfy φ *assuming* its environment behaves in a particular way.
- B may meet these assumptions.

Assume guarantee reasoning

Find some C such that:

- $A \parallel C \models \varphi$
- $B \preceq C$

Key observation: C could be *much smaller* than A and B .

Compositional verification

Assume finite-state systems A and B .

To check if $A \parallel B \models \varphi$.

- Assume for now that φ is a property only about A .
- A may not satisfy φ in a general environment.
 But satisfy φ *assuming* its environment behaves in a particular way.
- B may meet these assumptions.

Assume guarantee reasoning

Find some C such that:

- $A \parallel C \models \varphi$
- $B \preceq C$

Key observation: C could be *much smaller* than A and B .

Compositional verification

Assume finite-state systems A and B .

To check if $A \parallel B \models \varphi$.

- Assume for now that φ is a property only about A .
- A may not satisfy φ in a general environment.
But satisfy φ *assuming* its environment behaves in a particular way.
- B may meet these assumptions.

Assume guarantee reasoning

Find some C such that:

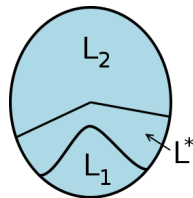
- $A \parallel C \models \varphi$
- $B \preceq C$

Key observation: C could be *much smaller* than A and B !

Compositional verification

It's an inclusion learning problem!

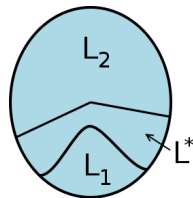
- L_1 is the language of B
- L_2 is the *most general environment* under which A satisfies φ
- L^* is the assumption-commitment automaton C we want to learn.
- Learning algorithm learns $L^* = C$
- Teacher answers queries using *separate* model-checking calls to A and B
- Works in practice in finding small automata C using which global system can be verified.



Compositional verification

It's an inclusion learning problem!

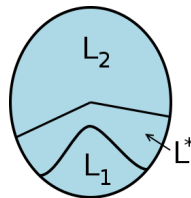
- L_1 is the language of B
- L_2 is the *most general environment* under which A satisfies φ
- L^* is the assumption-commitment automaton C we want to learn.
- Learning algorithm learns $L^* = C$
- Teacher answers queries using *separate* model-checking calls to A and B
- Works in practice in finding small automata C using which global system can be verified.



Compositional verification

It's an inclusion learning problem!

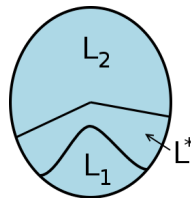
- L_1 is the language of B
 - L_2 is the *most general environment* under which A satisfies φ
 - L^* is the assumption-commitment automaton C we want to learn.
- Learning algorithm learns $L^* = C$
 - Teacher answers queries using *separate* model-checking calls to A and B
 - Works in practice in finding small automata C using which global system can be verified.



Compositional verification





It's an inclusion learning problem!

- L_1 is the language of B
- L_2 is the *most general environment* under which A satisfies φ
- L^* is the assumption-commitment automaton C we want to learn.
- Learning algorithm learns $L^* = C$
- Teacher answers queries using *separate* model-checking calls to A and B
- Works in practice in finding small automata C using which global system can be verified.



References for Compositional Verification

The NASA folks:

-  Barringer, Giannakopoulou, Pasareanu: Proof rules for automated compositional verification through learning ; [Spec. & Verif. of Component Based Systems, 2003](#) .
-  Cobleigh, Giannakopoulou, Pasareanu: Learning assumptions for compositional verification ; [TACAS 2003](#) .
-  Giannakopoulou, Pasareanu: Learning-based assume-guarantee verification ; [SPIN'05](#) .
-  Pasareanu, Giannakopoulou: Towards a compositional SPIN ; [SPIN'06](#) .

References for Compositional Verification

-  Alur, Madhusudan, Nam: Symbolic Compositional Verification by learning assumptions ; [CAV'05](#) .
-  Nam, Alur: Learning-Based Symbolic Assume-Guarantee Reasoning with Automatic Decomposition ; [ATVA'06](#) .
-  Chaki, Clarke, Sinha, Thati Automated assume-guarantee reasoning for simulation conformance ; [CAV 2005](#) .
-  Chaki, Clarke, Sharygina, Sinha Dynamic Component Substitutability Analysis ; [FM'05](#) .
-  Chaki, Sinha Assume-Guarantee Reasoning for Deadlocks ; [FMCAD'06](#) .

Interface Synthesis

Problem

Given a piece of code, find the intended way to use the code.

- Given A , a class of methods, find all sequences of method-calls that will *not* raise an exception.
- We expect the set of these sequences to be *simple*.
- Is a *game problem* between system and environment:
- Environment plays method-calls; system executes it
- Environment wins if no exception occurs
- Problem: Synthesize the maximal *winning strategy* for the environment.

Interface Synthesis

Problem

Given a piece of code, find the intended way to use the code.

- Given A , a class of methods, find all sequences of method-calls that will *not* raise an exception.
- We expect the set of these sequences to be *simple*.
- Is a *game problem* between system and environment:
- Environment plays method-calls; system executes it
- Environment wins if no exception occurs
- Problem: Synthesize the maximal *winning strategy* for the environment.

Interface Synthesis

Problem

Given a piece of code, find the intended way to use the code.

- Given A , a class of methods, find all sequences of method-calls that will *not* raise an exception.
- We expect the set of these sequences to be *simple*.
- Is a *game problem* between system and environment:
- Environment plays method-calls; system executes it
- Environment wins if no exception occurs
- Problem: Synthesize the maximal *winning strategy* for the environment.

Interface Synthesis

Problem

Given a piece of code, find the intended way to use the code.

- Given A , a class of methods, find all sequences of method-calls that will *not* raise an exception.
- We expect the set of these sequences to be *simple*.
- Is a *game problem* between system and environment:
 - Environment plays method-calls; system executes it
 - Environment wins if no exception occurs
 - Problem: Synthesize the maximal *winning strategy* for the environment.

Interface Synthesis

Problem

Given a piece of code, find the intended way to use the code.

- Given A , a class of methods, find all sequences of method-calls that will *not* raise an exception.
- We expect the set of these sequences to be *simple*.
- Is a *game problem* between system and environment:
- Environment plays method-calls; system executes it
- Environment wins if no exception occurs
- Problem: Synthesize the maximal *winning strategy* for the environment.

Interface Synthesis

Key Idea

Learn the winning strategy:

- Membership and subset queries can be answered using model-checking
- Superset queries are hard and answered using heuristics
- Works well in practice: mines simple interfaces from code



Alur, Cerny, Madhusudan, Nam: Synthesis of interface specifications for Java classes ; [POPL'05](#) .

Interface Synthesis

Key Idea

Learn the winning strategy:

- Membership and subset queries can be answered using model-checking
- Superset queries are hard and answered using heuristics
- Works well in practice: mines simple interfaces from code



Alur, Cerny, Madhusudan, Nam: Synthesis of interface specifications for Java classes ; [POPL'05](#) .

Learning the Reachable Set

Problem

For an infinite state system (say a FIFO system), can we learn the set of reachable states?

Key problem: Membership is hard!

- Idea: Learn states with witnesses
- Membership is solvable in the presence of a witness
- Witnesses need to be chosen carefully!

Learning the Reachable Set

Problem

For an infinite state system (say a FIFO system), can we learn the set of reachable states?

Key problem: Membership is hard!

- Idea: Learn states with witnesses
- Membership is solvable in the presence of a witness
- Witnesses need to be chosen carefully!

Learning the Reachable Set




Problem

For an infinite state system (say a FIFO system), can we learn the set of reachable states?

Key problem: Membership is hard!

- Idea: Learn states with witnesses
- Membership is solvable in the presence of a witness
- Witnesses need to be chosen carefully!

Learning the Reachable Set

-  Vardhan, Sen, Viswanathan, Agha: Learning to verify safety properties ; [ICFEM'04](#) .
-  Vardhan, Sen, Viswanathan, Agha: Actively learning to verify safety of FIFO automata ; [FSTTCS'04](#) .
-  Vardhan, Viswanathan: LEVER: A tool for learning based verification ; [CAV'06](#) .
-  Vardhan, Viswanathan: Learning to Verify Branching Time Properties ; [ASE'05](#) .

Black-box checking

- Model-check an unknown system
- Learn a model from it (using tests) and verify the model
- Crucial: No equivalence test
- But size of model must be known.
- Technically: Learning + Conformance testing



Peled, Vardi, Yannakakis: Black-box checking ;
[FORTE/PSTV'99](#) .



Groce, Peled, Yannakakis Adaptive Model Checking ; [TACAS'02](#)

Black-box checking

- Model-check an unknown system
- Learn a model from it (using tests) and verify the model
- Crucial: No equivalence test
- But size of model must be known.
- Technically: Learning + Conformance testing



Peled, Vardi, Yannakakis: Black-box checking ;
[FORTE/PSTV'99](#) .



Groce, Peled, Yannakakis Adaptive Model Checking ; [TACAS'02](#)

Black-box checking

- Model-check an unknown system
- Learn a model from it (using tests) and verify the model
- Crucial: No equivalence test
- But size of model must be known.
- Technically: Learning + Conformance testing



Peled, Vardi, Yannakakis: Black-box checking ;
[FORTE/PSTV'99](#) .



Groce, Peled, Yannakakis Adaptive Model Checking ; [TACAS'02](#)

Black-box checking

- Model-check an unknown system
- Learn a model from it (using tests) and verify the model
- Crucial: No equivalence test
- But size of model must be known.
- Technically: Learning + Conformance testing



Peled, Vardi, Yannakakis: Black-box checking ;
[FORTE/PSTV'99](#) .



Groce, Peled, Yannakakis Adaptive Model Checking ; [TACAS'02](#)

Black-box checking

- Model-check an unknown system
- Learn a model from it (using tests) and verify the model
- Crucial: No equivalence test
- But size of model must be known.
- Technically: Learning + Conformance testing



Peled, Vardi, Yannakakis: Black-box checking ;
[FORTE/PSTV'99](#) .



Groce, Peled, Yannakakis Adaptive Model Checking ; [TACAS'02](#)

.

Summary

- **Learning is a tool for Occam's razor**
- In verification, it can be used to *mine* simple concepts from code, which can then be used to verify systems.
- A theoretical tool that can formalize heuristical tricks to handle real-life examples.
- Learning regular languages can be achieved in poly-time and is the algorithm most exploited in verification.
- A completely different approach to verification with a promising future

Summary

- Learning is a tool for Occam's razor
- In verification, it can be used to *mine* simple concepts from code, which can then be used to verify systems.
- A theoretical tool that can formalize heuristical tricks to handle real-life examples.
- Learning regular languages can be achieved in poly-time and is the algorithm most exploited in verification.
- A completely different approach to verification with a promising future

Summary

- Learning is a tool for Occam's razor
- In verification, it can be used to *mine* simple concepts from code, which can then be used to verify systems.
- A theoretical tool that can formalize heuristical tricks to handle real-life examples.
- Learning regular languages can be achieved in poly-time and is the algorithm most exploited in verification.
- A completely different approach to verification with a promising future

Summary

- Learning is a tool for Occam's razor
- In verification, it can be used to *mine* simple concepts from code, which can then be used to verify systems.
- A theoretical tool that can formalize heuristical tricks to handle real-life examples.
- Learning regular languages can be achieved in poly-time and is the algorithm most exploited in verification.
- A completely different approach to verification with a promising future

Summary

- Learning is a tool for Occam's razor
- In verification, it can be used to *mine* simple concepts from code, which can then be used to verify systems.
- A theoretical tool that can formalize heuristical tricks to handle real-life examples.
- Learning regular languages can be achieved in poly-time and is the algorithm most exploited in verification.
- A completely different approach to verification with a promising future

Open Questions

- Theory has shied away from NP-hard problems
 - Can we leverage SAT to effectively learn concepts?
 - Can the inclusion learning problem be solved using SAT?
- We have completely ignored the PAC model of learning
 - Can we use randomization to effectively learn approximate models?
 - Can we use *passive learning* to build models?
- Very few concepts have been subject to learning
 - Can we learn *loop invariants*?
 - Can we learn *shape invariants of dynamic structures*?
 - Can we use learning instead of widening/narrowing/accelerating techniques?

Go forth and learn!

Open Questions

- Theory has shied away from NP-hard problems
 - Can we leverage SAT to effectively learn concepts?
 - Can the inclusion learning problem be solved using SAT?
- We have completely ignored the PAC model of learning
 - Can we use randomization to effectively learn approximate models?
 - Can we use *passive learning* to build models?
- Very few concepts have been subject to learning
 - Can we learn *loop invariants*?
 - Can we learn *shape invariants of dynamic structures*?
 - Can we use learning instead of widening/narrowing/accelerating techniques?

Go forth and learn!



Open Questions

- Theory has shied away from NP-hard problems
 - Can we leverage SAT to effectively learn concepts?
 - Can the inclusion learning problem be solved using SAT?
- We have completely ignored the PAC model of learning
 - Can we use randomization to effectively learn approximate models?
 - Can we use *passive learning* to build models?
- Very few concepts have been subject to learning
 - Can we learn *loop invariants*?
 - Can we learn *shape invariants of dynamic structures*?
 - Can we use learning instead of widening/narrowing/accelerating techniques?

Go forth and learn!



Open Questions

- Theory has shied away from NP-hard problems
 - Can we leverage SAT to effectively learn concepts?
 - Can the inclusion learning problem be solved using SAT?
- We have completely ignored the PAC model of learning
 - Can we use randomization to effectively learn approximate models?
 - Can we use *passive learning* to build models?
- Very few concepts have been subject to learning
 - Can we learn *loop invariants*?
 - Can we learn *shape invariants of dynamic structures*?
 - Can we use learning instead of widening/narrowing/accelerating techniques?

Go forth and learn!



Open Questions

- Theory has shied away from NP-hard problems
 - Can we leverage SAT to effectively learn concepts?
 - Can the inclusion learning problem be solved using SAT?
- We have completely ignored the PAC model of learning
 - Can we use randomization to effectively learn approximate models?
 - Can we use *passive learning* to build models?
- Very few concepts have been subject to learning
 - Can we learn *loop invariants*?
 - Can we learn *shape invariants of dynamic structures*?
 - Can we use learning instead of widening/narrowing/accelerating techniques?

Go forth and learn!