Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Mathématiques et Informatique**

Arrêté ministérial : 25 mai 2016

Présentée par

Irini-Eleftheria Mens

Thèse dirigée par **Oded Maler**,

préparée au sein du **Laboratoire VERIMAG**
dans **l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Learning Regular Languages over Large Alphabets

Apprentissage de langages réguliers sur des alphabets de grandes tailles

Thèse soutenue publiquement le **10 October 2017**,
devant le jury composé de :

Monsieur Oded Maler
DIRECTEUR DE RECHERCHE, CNRS DÉLÉGATION ALPES, Directeur de thèse
Madame Dana Angluin
PROFESSEURE, UNIVERSITÉ YALE A NEW HAVEN - USA, Rapporteur
Monsieur Peter Habermehl
MAÎTRE DE CONFÉRENCES, UNIVERSITÉ PARIS 7, Rapporteur
Monsieur Eric Gaussier
PROFESSEUR, UNIVERSITÉ GRENOBLE ALPES, Président
Monsieur Frits W. Vaandrager
PROFESSEUR, UNIV. RADBOUD DE NIMEGUE - PAYS-BAS, Examinateur
Monsieur Laurent Fribourg
DIRECTEUR DE RECHERCHE, CNRS DÉLÉGATION PARIS-VILLEJUIF, Examinateur

Abstract

Learning regular languages is a branch of machine learning, a domain which has proved useful in many areas, including artificial intelligence, neural networks, data mining, verification, etc. In addition, interest in languages defined over large and infinite alphabets has increased in recent years. Although many theories and properties generalize well from the finite case, learning such languages is not an easy task. As the existing methods for learning regular languages depend on the size of the alphabet, a straightforward generalization in this context is not possible.

In this thesis, we present a generic algorithmic scheme that can be used for learning languages defined over large or infinite alphabets, such as bounded subsets of $\mathbb{N}$ or $\mathbb{R}$ or Boolean vectors of high dimensions. We restrict ourselves to the class of languages accepted by deterministic symbolic automata that use predicates to label transitions, forming a finite partition of the alphabet for every state.

Our learning algorithm, an adaptation of Angluin's L^*, combines standard automaton learning by state characterization, with the learning of the static predicates that define the alphabet partitions. We use the online learning scheme, where two types of queries provide the necessary information about the target language. The first type, membership queries, answer whether a given word belongs or not to the target. The second, equivalence queries, check whether a conjectured automaton accepts the target language and provide a counter-example otherwise.

We study language learning over large or infinite alphabets within a general framework but our aim is to provide solutions for particular concrete instances. For this, we focus on the two main aspects of the problem. Initially, we assume that equivalence queries always provide a counter-example which is minimal in the length-lexicographic order when the conjecture automaton is incorrect. Then, we drop this "strong" equivalence oracle and replace it by a more realistic assumption, where equivalence is approximated by testing queries, which use sampling on the set of words. Such queries are not guaranteed to find counter-examples and certainly not minimal ones. In this case, we obtain the weaker notion of PAC (probably approximately correct) learnability and learn an approximation of the target language. All proposed algorithms have been implemented and their performance, as a function of automaton and alphabet size, has been empirically evaluated.

Résumé

L'apprentissage de langages réguliers est une branche de l'apprentissage automatique qui s'est révélée utile dans de nombreux domaines tels que l'intelligence artificielle, les réseaux de neurones, l'exploration de données, la vérification, etc. De plus, l'intérêt dans les langages définis sur des alphabets infinis ou de grande taille s'est accru au fil des années. Même si plusieurs propriétés et théories se généralisent à partir du cas fini, l'apprentissage de tels langages est une tâche difficile. En effet, dans ce contexte, l'application naïve des algorithmes d'apprentissage traditionnel n'est pas possible.

Dans cette thèse, nous présentons un schéma algorithmique général pour l'apprentissage de langages définis sur des alphabets infinis ou de grande taille, comme par exemple des sous-ensembles bornés de $\mathbb{N}$ or $\mathbb{R}$ ou des vecteurs booléens de grandes dimensions. Nous nous restreignons aux classes de langages qui sont acceptés par des automates déterministes symboliques utilisant des prédicats pour définir les transitions, construisant ainsi une partition finie de l'alphabet pour chaque état.

Notre algorithme d'apprentissage, qui est une adaptation du L^* d'Angluin, combine l'apprentissage classique d'un automate par la caractérisation de ses états, avec l'apprentissage de prédicats statiques définissant les partitions de l'alphabet. Nous utilisons l'apprentissage incrémental avec la propriété que deux types de requêtes fournissent une information suffisante sur le langage cible. Les requêtes du premier type sont les requêtes d'appartenance, qui permettent de savoir si un mot proposé appartient ou non au langage cible. Les requêtes du second type sont les requêtes d'équivalence, qui vérifient si un automate proposé accepte le langage cible; dans le cas contraire, un contre-exemple est renvoyé.

Nous étudions l'apprentissage de langages définis sur des alphabets infinis ou de grande tailles dans un cadre théorique et général, mais notre objectif est de proposer des solutions concrètes pour un certain nombre de cas particuliers. Ensuite, nous nous intéressons aux deux principaux aspects du problème. Dans un premier temps, nous supposerons que les requêtes d'équivalence renvoient toujours un contre-exemple minimal pour un ordre de longueur-lexicographique quand l'automate proposé est incorrect. Puis dans un second temps, nous relâchons cette hypothèse forte d'un oracle d'équivalence, et nous la remplaçons avec une hy-

pothèse plus réaliste où l'équivalence est approchée par un test sur les requêtes qui utilisent un échantillonnage sur l'ensemble des mots. Dans ce dernier cas, ce type de requêtes ne garantit pas l'obtention de contre-exemples, et par conséquent de contre-exemples minimaux. Nous obtenons alors une notion plus faible d'apprentissage PAC (*Probably Approximately Correct*), permettant l'apprentissage d'une approximation du langage cible. Tout les algorithmes ont été implémentés, et leurs performances, en terme de construction d'automate et de taille d'alphabet, ont été évaluées empiriquement.

Contents

List of Algorithms

Notation

General notation	
$\mathbb{N}$	the set of *natural* numbers
$\mathbb{Q}$	the set of *rational* numbers
$\mathbb{R}$	the set of *real* numbers
$\mathbb{B}^n$	the set $\{0,1\}^n$, where $n \in \mathbb{N}$
$\lvert A \rvert$	the cardinality of a finite set A
$\max(A)$	the *maximum* value in a set A
$\min(A)$	the *minimum* value in a set A
$\arg\max_x\{f(x)\}$	the value x which maximizes the function f
$A \oplus B$	the symmetric difference of two sets
$A \uplus B$	the union of pairwise disjoint sets

Notation used in this thesis	
Σ	concrete alphabet (finite or infinite)
$\boldsymbol{\Sigma}$	symbolic alphabet, a finite set of symbols
$\boldsymbol{a}, \boldsymbol{b}, \ldots$	symbolic letters (or symbols)
$[\![\boldsymbol{a}]\!]$	semantics of symbol $\boldsymbol{a}$
$\mathcal{V}(\boldsymbol{a})$	set of symbols with adjacent semantics to $[\![\boldsymbol{a}]\!]$

Abbreviations	
DFA	Deterministic Finite Automaton
BDT	Binary Decision Tree
DNF	Disjunctive Normal Form
PBNF	Pseudo-Boolean Normal Form
MQ	Membership Query
EQ	Equivalence Query

Chapter 1
Introduction

1.1 Synopsis

In its most general and abstract form, the problem of machine learning can be phrased as follows. Given a function $f : X \to Y$ and a sample S consisting of $(x, f(x))$ pairs, construct a function f' that agrees, exactly or approximately, with f on the sample. In many cases, where X is a large domain and Y is a small one, the function f is viewed as a classifier, i.e., a rule that classifies objects in X into one of the finitely many categories in Y. In other words, X is partitioned into finitely many subsets that share some property. In the case where $Y = \mathbb{B} = \{0, 1\}$, f is the characteristic function of a set $L \subseteq X$, and the sample consists of positive ($f(x) = 1$) and negative ($f(x) = 0$) examples.

Formal languages are subsets of Σ^*, that is, the (infinite) set of all finite sequences (words) over an alphabet Σ. The major topic of this thesis concerns the problem of language identification, an instance of machine learning where one wants to infer the characteristic function of a target language $L \subseteq \Sigma^*$. More precisely, we work within the framework of active learning, that is, the learning algorithm can select, in an adaptive manner, the words for which membership in L is queried. Our starting point is the L^* algorithm of Angluin [Ang87], a well-known algorithm for learning regular languages based on membership queries, which also makes use of counter-examples provided by a helpful teacher.

The essence of automaton learning algorithms, such as L^*, is to organize the information on word membership in a two-dimensional observation table whose rows indicate words (prefixes) that lead to states in the minimal automaton for L and its columns represent suffixes whose acceptance status after the prefix characterizes the states based on the Nerode right-congruence relation [Ner58]. Depending on the learning protocol, the number of queries needed to learn a language L is poly-

nomial or exponential in the number of states of the minimal automaton accepting L. It is linear in the size of the alphabet Σ [BR05].

We extend this algorithm to work in situations where the size of the alphabet is prohibitively large or even infinite, as in the case of $\mathbb{B}^n$ or $\mathbb{R}$, and the accepting automaton cannot be represented traditionally by writing down all the letters that label its transitions. We represent languages on such large alphabets using symbolic automata with a modest number of transitions, each labeled by a predicate on the input alphabet. In each state, these predicates define a partition of the alphabet resulting in a deterministic automaton.

We adapt the automaton learning framework to this setting in a generic way, which does not make strong assumptions on the type of the large alphabet. The idea is to associate with each state a symbolic alphabet whose letters label the transitions outgoing from that state. Symbolic sequences serve as access sequences to states (rows in the observation table) in the same manner as concrete sequences do for the L^* algorithm. The concrete semantics of the symbolic letters are defined via predicates that map them to sets of concrete letters.

The crucial difference from the small alphabet setting is that, rather than the entire alphabet, only a small sample (evidence) of the presumed semantics of each symbolic letter is tested with membership queries. As a consequence, this semantics is not guaranteed to be correct and it might be that the partition boundaries are imprecise, or even that some symbolic letter (and its corresponding transition) has not been discovered yet.

To explain this with a concrete example, let the alphabet be $[0, 10)$ and suppose that at some point in time we associate two symbolic letters $\boldsymbol{a}$ and $\boldsymbol{b}$ with a state q_1. Consider their respective semantics to be $[\![\boldsymbol{a}]\!] = [0, 5)$ and $[\![\boldsymbol{b}]\!] = [5, 10)$ with transitions $\delta(q_1, \boldsymbol{a}) = q_2$ and $\delta(q_1, \boldsymbol{b}) = q_3$, respectively. At some later point in time, we may discover that some concrete letters in $[4, 6)$ behave differently. We introduce then a new symbol c, modify the semantics to $[\![\boldsymbol{a}]\!] = [0, 4)$, $[\![\boldsymbol{c}]\!] = [4, 6)$, and $[\![\boldsymbol{b}]\!] = [6, 10)$, and add the transition $\delta(q_1, \boldsymbol{c}) = q_1$. As we will see, the process of learning over large alphabets interleaves two activities: the discovery of new states, as in standard automaton learning, and the modification of the partition boundaries in each state. The latter part is done for each state using sampling and static machine learning techniques, depending on the nature of the alphabet.

Algorithm L^*, in addition to membership queries, has a helpful teacher that, whenever the learner conjectures an automaton, either confirms it or presents a counter-example. Such a counter-example simplifies the discovery of new states. One version of our algorithm uses such a helpful teacher and, moreover, assumes the counter-example to be minimal in the lexicographic order (or partial order) induced by the domain. This assumption simplifies the modification of the partition boundaries due to counter-examples, allowing exact learning of the partitions.

In another version of the algorithm, we replace this teacher by a more realistic equivalence checking technique that uses random sampling and membership queries. This may or may not lead to counter-examples. We develop a method for analyzing these (non-minimal) counter-examples and see whether they lead to a discovery of a new state or to a modification in the alphabet partition for an existing state. In this setting, a certain convergence to the exact target language cannot be guaranteed and for this, it is replaced by the probably approximately correct (PAC) condition.

We consider, then, the following types of alphabets and partitions/predicates.

1. Most of the work is done for one-dimensional numerical domains, such as bounded sub-intervals of $\mathbb{R}$ or $\mathbb{N}$, partitioned into finitely many intervals. To this domain we apply algorithms with and without a helpful teacher that provides counter-examples to conjectures;
2. For the domain $\mathbb{B}^n$ of Boolean vectors, we consider partitions expressed using decision trees and apply a teacher-free algorithm;
3. For multi-dimensional domains, such as bounded subsets of $\mathbb{R}^n$ or $\mathbb{N}^n$, we use monotone partitions that generalize intervals to partially-ordered sets and apply an algorithm that uses a helpful teacher.

1.2 Motivation

Finite automata are among the corner stones of Computer Science. From a practical point of view, they are used routinely in various domains ranging from syntactic analysis, design of user interfaces or administrative procedures to implementation of digital hardware and verification of software and hardware protocols. Regular languages admit a very nice, clean and comprehensive theory where different formalisms such as automata, logic, regular expressions, semigroups, and grammars are shown to be equivalent.

One weakness, however, of the classical theory of regular languages and automata is that it mainly deals with alphabets that are rather "thin" and "flat", that is, sets that are small and do not admit any additional structure. In many applications, however, alphabets are large and structured. In hardware verification, for example, behaviors are sequences of states and inputs ranging over valuations of Boolean state variables that give rise to exponentially large alphabets, treated symbolically using BDDs and other logical formalisms. As another motivation, one can consider the verification of continues or hybrid systems against specifications written in a formalism such as signal temporal logic (STL) [MN04, MNP08]. Automata over numerical alphabets, admitting an order or partially-order relation, can define the semantics of such requirements.

Symbolic automata [VBDM10], like the ones we use here, give a more succinct representation for languages over large finite alphabets and can also represent languages over infinite alphabets such as $\mathbb{N}$, $\mathbb{R}$, or $\mathbb{R}^n$. Other applications that benefit from the succinct representation of symbolic automata and the usage of large alphabets include, among other, natural language processing [VNG01], regular expressions in the context of static and dynamic program analysis [VDHT10], constraint solving and symbolic analysis with SMT solvers [VBDM10] or security analysis of string sanitizers [Vea13].

Finally, learning automata from examples provides a crisp characterization of what a *state* in a dynamical system is in terms of observable input-output behavior. In contrast to classical machine learning, language inference aims to build an automaton model that will be used later for further analysis rather than simple characterization and prediction of some unobserved behavior of a future input. State system identification and language inference admit a variety of applications ranging from linguistics and biology to computer science. It is also widely used in software analysis and verification. Within software analysis, it is applied more specifically, in program analysis [WBAH08], software testing [AJU10, MN10], security testing [CSS+10], dynamic testing [RMSM09], integration testing [Sha08], and black-box components testing [Nie03, SL07, Sha08].

1.3 Outline

The rest of the thesis is organized as follows.

Preliminaries.

Chapter 2 provides the basic definitions and notations for the theory of regular languages and automata. Moreover, it presents different types of domain partitioning as well as static machine learning algorithms that can be used to learn such partitions from given sample points.

Chapter 3 provides a summary of learning algorithms over small alphabets. The state of the art is followed by a presentation of the L^* algorithm and a discussion on the treatment of counter-examples.

In Chapter 4 one can find a full presentation of symbolic automata in general, accompanied with special cases of domain-specific symbolic automata.

Contributions.

Chapter 5 first provides the basic elements that are used in the symbolic learning algorithm and then presents the algorithm without making any assumptions neither on the nature of the teacher or on the alphabet. Moreover in this chapter, one can find a comparison to related work.

Chapter 6 provides an adaptation of the more general symbolic algorithm presented in Chapter 5 to the case of a helpful teacher that always provides a counter-example when there is one, and in addition, this is a minimal one in the case of ordered alphabets. We study the cases of numerical alphabets such as $\mathbb{N}$, $\mathbb{R}$ and $\mathbb{R}^n$ [MM14, MM15].

Chapter 7 describes the symbolic algorithm in the context of a teacher who can only answer membership queries and thus counter-examples when provided, are not necessarily minimal. We focus on the cases of one-dimensional numerical domains and Boolean alphabets, where the partitions are inferred using static classification learning algorithms [MM17].

Chapter 8 provides some theoretical evaluation of the algorithm in terms of termination and the trade-off between accuracy and complexity.

Chapter 9 presents case studies that we have tested to validate the algorithms, evaluate their performance and compare them, when possible, with concrete learning algorithms.

Conclusion.

Chapter 10 provides the conclusion and future directions.

CHAPTER 2
Preliminaries

In this chapter, we provide the basic definitions and notations that are used throughout this thesis. Section 2.1 succinctly provides the definitions and notations concerning regular languages and finite state automata; for a full reference on the topic, the reader may consult the books [LP97, HMU06, Sip06]. In Section 2.2, we explain what a domain partition into classes is, and present some specific cases that are used in the subsequent chapters. Finally in Section 2.3, we discuss some learning methods that can be applied to identify an unknown partition.

2.1 Regular Languages and Automata

An *alphabet* Σ is a non-empty set whose elements are called *letters*. A *word* (or *string*) is a finite sequence of letters, i.e., $w = a_1 \dots a_n$, chosen from the alphabet Σ; the number of letters in this sequence determines the *length* of the word, i.e., $|w| = n$. The sequence of zero length is called the *empty word* and it is denoted by ε. The set of all words over an alphabet Σ is denoted by Σ^*.

Let $w = a_1 \dots a_n$ and $u = b_1 \dots b_m$ be two words. The two words w and u are said to be *equal* when they have equal length and the same letter at each position, i.e., $w = u$, if and only if $n = m$ and $a_i = b_i$ for all $i = 1, \dots, n$. The *concatenation* of two words w and u is the sequence $w \cdot u = a_1 \dots a_n b_1 \dots b_m$. A word u is a *prefix* (*resp.* *suffix*) of w, if there exists $v \in \Sigma^*$ such that $w = u \cdot v$ (*resp.* $w = v \cdot u$).

Let $\leq$ be an order relation over Σ. This can be naturally lifted to an order relation over Σ^*. Some of the most interesting order relations defined on the set of words are the following.

1. The *prefix order*, $(\Sigma^*, \leq_{pr})$, where $u \leq_{pr} w$ if and only if u is a prefix of w.
2. The *lexicographic order*, $(\Sigma^*, \leq_{lex})$, where $u \leq_{lex} w$ if and only if either

$u \leq_{pr} w$, or $u = v \cdot a \cdot v'$ and $w = v \cdot b \cdot v''$ for some $v, v', v'' \in \Sigma^*$, $a, b \in \Sigma$, where $a \leq b$ and $a \neq b$.

3. The *length-lexicographic order*, $(\Sigma^*, \leq_{ll})$, where $u \leq_{ll} w$ if and only if either $|u| < |w|$, or $|u| = |w|$ and $u \leq_{lex} w$.

Note that, if $(\Sigma, \leq)$ defines a total order, then the orders $(\Sigma^*, \leq_{lex})$ and $(\Sigma^*, \leq_{ll})$ are also total.

Any subset of words L chosen from Σ^* is called a *language* over Σ. To each language L we can associate a *characteristic function* $f : \Sigma^* \to \{+, -\}$[1] such that $f(w) = +$ if the word w belongs to L and $f(w) = -$, otherwise. Moreover, with every $s \in \Sigma^*$ we can associate a *residual characteristic function* defined as $f_s(w) = f(s \cdot w)$.

Several operations can be defined on languages. The *Boolean operations* include the *union*, *intersection*, *complementation*, and *difference*. The *symmetric difference* is also commonly used. Formally, let L and L' be two languages over Σ, these operations are defined, respectively, as

$$
\begin{aligned}
L \cup L' &= \{w \in \Sigma^* : w \in L \text{ or } w \in L'\} \\
L \cap L' &= \{w \in \Sigma^* : w \in L \text{ and } w \in L'\} \\
\bar{L} &= \{w \in \Sigma^* : w \notin L\} \\
L \setminus L' &= \{w \in \Sigma^* : w \in L \text{ and } w \notin L'\} \\
L \oplus L' &= (L \setminus L') \cup (L' \setminus L)
\end{aligned}
$$

The *concatenation* (or *product*) of two languages L and L' is the set of words that can be formed by taking any word in L and concatenating it with any word in L', that is,

$$L \cdot L' = \{w \cdot w' : w \in L \text{ and } w' \in L'\}.$$

The *power* of a language L is defined as $L^{k+1} = L^k \cdot L = L \cdot L^k$ for all $k \geq 1$ where $L^0 = \{\varepsilon\}$. The *star* operation is the union of all the powers of a language L and it is defined as

$$L^* = \bigcup_{k \in \mathbb{N}} L^k.$$

The *left quotient* (*resp. right quotient*) of L is the language

$$u^{-1}L = \{v \in \Sigma^* : uv \in L\} \quad (\textit{resp. } Lu^{-1} = \{v \in \Sigma^* : vu \in L\}).$$

A language $L \subseteq \Sigma^*$ is *prefix-closed* (*resp. suffix-closed*) if and only if for all $w, u \in \Sigma^*$, $w \cdot u \in L$ implies $w \in L$ (*resp.* $u \in L$).

[1] The notation $\{1, 0\}$ or $\{True, False\}$ is also commonly used.

A language $L \subseteq \Sigma^*$ is a *regular language* if it can be recursively constructed by applying the union, concatenation and star operations finitely many times. By default, the empty language, the language $\{\varepsilon\}$, and the one letter languages $\{a\}$ for all $a \in \Sigma$, are regular languages. Regular languages over finite alphabets can be represented by deterministic finite automata.

A *deterministic finite automaton* (DFA) over Σ is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$, where Σ is a finite alphabet, Q is a non-empty finite set of *states*, $q_0 \in Q$ is the *initial* state, $\delta : Q \times \Sigma \to Q$ is a *transition function*, and $F \subseteq Q$ is a set of *final* (or *accepting*) states.

The transition function δ can be extended to $\delta : Q \times \Sigma^* \to Q$, where $\delta(q, \varepsilon) = q$ and $\delta(q, u \cdot a) = \delta(\delta(q, u), a)$ for $q \in Q$, $a \in \Sigma$ and $u \in \Sigma^*$. A word $w \in \Sigma^*$ is *accepted* by $\mathcal{A}$ if $\delta(q_0, w) \in F$, otherwise w is *rejected*. The *language recognized* (or *accepted*) by $\mathcal{A}$, denoted by $L(\mathcal{A})$, consists of the set of all accepted words.

A language $L \subseteq \Sigma^*$ is *recognizable* if there exists an automaton $\mathcal{A}$ such that $L = L(\mathcal{A})$. Let $REC(\Sigma)$ denote the set of all recognizable languages over the alphabet Σ; this set coincides with the set of regular languages.

Two automata that recognize the same language are said to be *equivalent*. Among all automata that are equivalent there exists a *minimal* automaton, also known as the *canonical* automaton, which is minimal in the number of states and unique up to isomorphism.

Let $L \subseteq \Sigma^*$ be a language. This induces an equivalence relation $\sim_L$ on Σ^*, the *Nerode equivalence relation*, where

$$s \sim_L r \text{ if and only if } f_s = f_r \tag{2.1}$$

Intuitively, two words are Nerode equivalent whenever by appending the same string to s and r, the resulting two strings are either both in L or both not in L. The relation $\sim_L$ is a right congruence satisfying $s \sim_L r \to s \cdot a \sim_L r \cdot a$ and is known as the *syntactic congruence* associated with L. Its equivalence classes correspond to the states of the minimal automaton that accepts L. The following theorem is part of what is known as the Myhill-Nerode Theorem

Theorem 2.1 (Nerode [Ner58])**.** *Let $L \subseteq \Sigma^*$ be a language, and let $\sim_L$ be an equivalence relation as defined in (2.1). Then, L is a regular language if and only if $\sim_L$ is of finite index.*

As a result, for any regular language $L \subseteq \Sigma^*$, one can find an automaton that is minimal using the syntactic congruence $\sim_L$. Based on Theorem 2.1, this automaton can be defined as $\mathcal{A}_M = (\Sigma, Q, q_0, \delta, F)$, where the set of states Q is the set of equivalence classes, i.e, $Q = \Sigma^* / \sim_L$; the initial state is the class that contains the empty word, i.e., $q_0 = [\varepsilon]$; the transition function is defined as $\delta([u], a) = [u \cdot a]$

for all $a \in \Sigma$; and, the final states are all those classes that contain accepted words, $F = \{[u] : u \cdot \varepsilon \in L\}$.

According to this, for an automaton to be considered minimal, all pairs of states in it should be distinguishable and thus not be equivalent. Two states p, q are distinguishable if there exists a word w such that $\delta(p, w) \in F$ and $\delta(q, w) \notin F$ (or $\delta(p, w) \notin F$ and $\delta(q, w) \in F$).

As the minimal representation is useful in many areas, several minimization algorithms have been proposed. One can refer to [Moo56, Brz62, Hop71] for the most well known automaton minimization algorithms. The Nerode congruence and the identification of minimal automata play a crucial role in most automata learning algorithms since [Gol72]. We discuss further this topic in Chapter 3.

2.2 Sets and Partitions

Let Σ be a finite or an infinite alphabet. A representation of Σ as the finite union of a family of pairwise disjoint subsets is called a *partition of* Σ *into classes*. This representation plays an important role in many problems [KF70]. Such partition is usually based on certain criteria that allow the assignment of elements from Σ to one class or another.

Let $\mathcal{C} = \{\boldsymbol{a}_1, \cdots, \boldsymbol{a}_k\}$ be a finite set of *classes* (or *labels*). A partition of Σ into classes can be defined as a mapping $\psi : \Sigma \to \mathcal{C}$ that associates each letter in Σ with a class in $\mathcal{C}$. Such a mapping is known as a *classification function*, and the tuple (Σ, ψ) is often called a *labeled alphabet*. The Σ*-semantics* of a label $\boldsymbol{a} \in \mathcal{C}$ denotes the set of elements from Σ that are associated to the same class $\boldsymbol{a}$, i.e., $[\![\boldsymbol{a}]\!] = \{a \in \Sigma : \psi(a) = \boldsymbol{a}\}$.

In the following, we provide some examples of such partitions. For readability, the alphabet Σ is presented in these examples as a subset of $\mathbb{N}$, $\mathbb{R}$, $\mathbb{R}^2$ or $\mathbb{B}^4$, but one should consider that Σ can be any subset of $\mathbb{N}^n$, $\mathbb{R}^n$ and $\mathbb{B}^n$, where $n \in \mathbb{N}$.

Intervals

Let Σ be a subset of the reals that admits the usual order relation $\leq$. We consider as an *interval* any connected segment[2] of $\mathbb{R}$. Let $a, b \in \mathbb{R}$ be the endpoints of an interval, we can determine the following interval types:

- *open* interval: $(a, b) = \{x \in \mathbb{R} : a < x < b\}$,
- *closed* interval: $[a, b] = \{x \in \mathbb{R} : a \leq x \leq b\}$,
- *closed-open* interval: $[a, b) = \{x \in \mathbb{R} : a \leq x < b\}$, and

[2] A *connected segment* of $\mathbb{R}$ is a non-empty subset X of $\mathbb{R}$ such that for all $a, b \in X$ and $c \in \mathbb{R}$, $a \leq c \leq b$ implies $c \in X$.

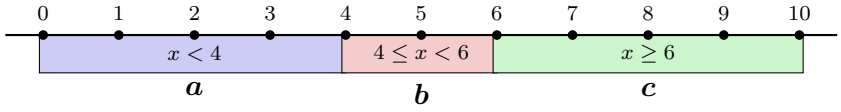


Figure 2.1: Partition of $\Sigma = [0, 10] \subseteq \mathbb{R}$ (or $\mathbb{N}$) into three intervals.

– *open-closed* interval: $(a, b] = \{x \in \mathbb{R} : a < x \leq b\}$.

An interval is empty when $b < a$ (or $b \leq a$ when the interval is not closed). By abuse of notation, we use the interval notation also for other numerical domains such as $\mathbb{N}$, e.g., $[a, b) = \{x \in \mathbb{N} : a \leq x < b\}$. An example of a partition of $\Sigma = [0, 10]$ into intervals is shown in Figure 2.1. In this example, Σ is partitioned into three intervals, each designated to a class from $\mathcal{C} = \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\}$. The classification function is given by

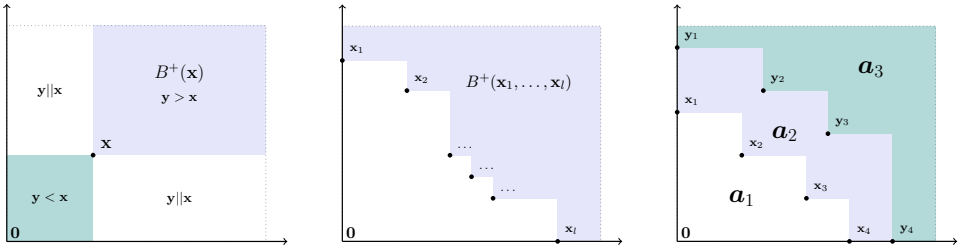
$$\psi(x) = \begin{cases} \boldsymbol{a}, & \text{for } x \in [0, 4) \\ \boldsymbol{b}, & \text{for } x \in [4, 6) \\ \boldsymbol{c}, & \text{for } x \in [6, 10] \end{cases} .$$

For a given partition of Σ, distinct classes with adjacent semantics can be considered as neighbors. Formally, a class $\boldsymbol{b} \in \mathcal{C}$ is a *right* (*resp. left*) *adjacent* to class $\boldsymbol{a} \in \mathcal{C}$ ($\boldsymbol{a} \neq \boldsymbol{b}$), if $a \in [\![\boldsymbol{a}]\!] \wedge c \in [\![\boldsymbol{b}]\!]$ (*resp.* $a \in [\![\boldsymbol{b}]\!] \wedge c \in [\![\boldsymbol{a}]\!]$) implies $b \in [\![\boldsymbol{a}]\!] \cup [\![\boldsymbol{b}]\!]$, for any three letters $a, b, c \in \Sigma$ such that $a < b < c$. The set of all right and left adjacent classes to $\boldsymbol{a} \in \Sigma$ forms the *neighborhood* of $\boldsymbol{a}$, denoted by $\mathcal{V}(\boldsymbol{a})$. Intuitively, the neighborhood of $\boldsymbol{a}$ consists of all those classes $\boldsymbol{b} \in \mathcal{C} \setminus \{\boldsymbol{a}\}$ with which $\boldsymbol{a}$ shares the same boundaries in their semantics. Referring to the example of Figure 2.1, class $\boldsymbol{b}$ is a right adjacent to $\boldsymbol{a}$ and a left adjacent to $\boldsymbol{c}$, but $\boldsymbol{a}$ and $\boldsymbol{c}$ are not adjacent. The neighborhoods of $\boldsymbol{a}$ and $\boldsymbol{b}$ are $\mathcal{V}(\boldsymbol{a}) = \{\boldsymbol{b}\}$ and $\mathcal{V}(\boldsymbol{b}) = \{\boldsymbol{a}, \boldsymbol{c}\}$, respectively.

Monotone Partitions

Let Σ be a partially-ordered alphabet of the form $\Sigma = X^n$, where X is a totally-ordered set such as an interval $[0, k) \subseteq \mathbb{R}$. Letters of Σ are n-tuples of the form $\mathbf{x} = (x_1, \ldots, x_n)$ with minimal element $\mathbf{0} = (0, \ldots, 0)$. The usual partial order on this set is defined as $\mathbf{x} \leq \mathbf{y}$ if and only if $x_i \leq y_i$ for all $i = 1, \ldots, n$. When $\mathbf{x} \leq \mathbf{y}$ and $x_i < y_i$ for some i, the inequality is strict, denoted by $\mathbf{x} < \mathbf{y}$, and we say then that $\mathbf{x}$ *dominates* $\mathbf{y}$. Two elements are *incomparable*, denoted by $\mathbf{x}||\mathbf{y}$, if $x_i < y_i$ and $x_j > y_j$ for some i and j.

For partially-ordered sets, a natural extension of the partition of an ordered set into intervals is a *monotone partition*, where for each partition block P there are



(a) (b) (c)

Figure 2.2: (a) Backward and forward cone for $\mathbf{x}$, (b) union of cones, (c) an alphabet partition into three classes $\mathcal{C} = \{\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3\}$.

no three points such that $\mathbf{x} < \mathbf{y} < \mathbf{z}$, $\mathbf{x}, \mathbf{z} \in P$, and $\mathbf{y} \notin P$. We will see in the following that a monotone partition with a finite number of partition blocks can be represented by a finite set of points.

A *forward cone* $B^+(\mathbf{x}) \subseteq \Sigma$ is the set of all points dominated by a point $\mathbf{x} \in \Sigma$, see Figure 2.2a. For a set of points $F = \{\mathbf{x}_1, \ldots, \mathbf{x}_l\}$, this cone can be extended to $B^+(F) = B^+(\mathbf{x}_1) \cup \ldots \cup B^+(\mathbf{x}_l)$, as shown in Figure 2.2b. When $\mathbf{x}_1, \ldots, \mathbf{x}_l$ are mutually incomparable, they constitute the set of minimal elements of F.

From a family of sets of points $\mathcal{F} = \{F_0, \ldots, F_{m-1}\}$, which satisfies $F_0 = \{\mathbf{0}\}$ and for every i: 1) $\forall \mathbf{y} \in F_i$, $\exists \mathbf{x} \in F_{i-1}$ such that $\mathbf{x} < \mathbf{y}$, and 2) $\forall \mathbf{y} \in F_i$, $\forall \mathbf{x} \in F_{i-1}$, $\mathbf{y} \not< \mathbf{x}$; one can define a monotone partition of the form $\mathcal{P} = \{P_1, \ldots, P_{m-1}\}$, where $P_i = B^+(F_{i-1}) - B^+(F_i)$.

An example of such a partition appears in Figure 2.2c. The partition is defined by the family of sets of incomparable points $\mathcal{F} = \{F_0, F_1, F_2\}$, where $F_0 = \{\mathbf{0}\}$, $F_1 = \{x_1, \ldots, x_4\}$, and $F_2 = \{y_1, \ldots, y_4\}$. The classification function can be given by $\psi(\mathbf{x}) = \boldsymbol{a}_i$ for all $\mathbf{x} \in P_i$.

The extension of the notions of adjacent classes and neighbors to the case of partially-ordered sets is straightforward. Hence in the example, the label $\boldsymbol{a}_1$ is a left adjacent to $\boldsymbol{a}_2$, but is not an adjacent to $\boldsymbol{a}_3$, and for the labels $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ the neighborhoods are $\mathcal{V}(\boldsymbol{a}_1) = \{\boldsymbol{a}_2\}$ and $\mathcal{V}(\boldsymbol{a}_2) = \{\boldsymbol{a}_1, \boldsymbol{a}_3\}$, respectively.

Partitions of the Boolean Cube

Let $\Sigma = \mathbb{B}^n$ be the n-dimension Boolean hyper-cube, where letters are boolean vectors and can be accessed by the Boolean variables $X = \{x_1, \ldots, x_n\}$. A literal is either a Boolean variable x, or its negation $\bar{x}$. A *sub-cube* (or *term*) of $\mathbb{B}^n$ is a conjunction of literals. We denote a sub-cube by $\phi_k = \prod_{i \in A_k} x_i \cdot \prod_{j \in B_k} \bar{x}_j$ where A_k, B_k are disjoint sets of indices from $\{1, \ldots, n\}$.

A possible way to partition a Boolean cube into classes is to partition it into sub-cubes, not necessarily of equal size, and associate each sub-cube with a class.

A *classification function* for the Boolean cube is thus a *pseudo-boolean function*[3] $\psi : \mathbb{B}^n \to \mathcal{C}$. Such a function can be represented in several ways; we either use a *pseudo-boolean normal form* or a *binary decision tree* to represent it, but in the case where the number of variables n is small, we often use a *Karnaugh map* representation as it offers a better visualization of the partition.

A *pseudo-boolean normal form* (PBNF), an analogous of a disjunctive normal form (DNF) for Boolean functions, is defined as:

$$\psi = \sum_{k=1}^{m} \boldsymbol{a}_k \Big(\prod_{i \in A_k} x_i \Big) \Big(\prod_{j \in B_k} \bar{x}_j \Big) = \sum_{k=1}^{m} \boldsymbol{a}_k \cdot \phi_k \tag{2.2}$$

where the coefficients $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m$ are labels from $\mathcal{C}$ and A_k, B_k are sets of indices such that i) $A_k, B_k \subseteq \{1, \ldots, n\}$, ii) $A_k \cap B_k = \emptyset$, iii) $A_k \cup B_k \neq \emptyset$, and iv) $(A_k \cap B_l) \cup (A_l \cap B_k) \neq \emptyset$ for $k, l = 1, 2, \ldots, m$, $k \neq l$.

Each one of the four conditions is required for the function ψ to be well defined and to form a partition of Σ. More precisely, we let $A_k \cup B_k \neq \emptyset$ to avert redundant terms in ψ; the function $\sum_k (\prod_{i \in A_k} x_i)(\prod_{j \in B_k} \bar{x}_j)$ is a DNF only when $A_k \cap B_k = \emptyset$; and moreover, it is orthogonal when no two terms intersect, i.e., $(A_k \cap B_l) \cup (A_l \cap B_k) \neq \emptyset$ for $k \neq l$. Orthogonality guarantees that ψ partitions Σ. The set of all letters $a \in \Sigma$ that model ϕ_k form the Σ-semantics of the class labeled by $\boldsymbol{a}_k$, i.e., $[\![\boldsymbol{a}_k]\!] = [\![\phi_k]\!] = \{a \in \Sigma : \phi_k(a)\}$.

A *binary decision tree* (BDT) (or *classification tree*) takes a letter as an input and, based on its attributes, assigns it to a class. A BDT is a tree T which at each node either tests some variable of the given input, or provides a class label. A BDT can hence be seen as a binary tree[4] T equipped with a mapping $d : \mathcal{P}os(T) \to X \uplus \mathcal{C}$, where $\mathcal{P}os(T)$ is the set of nodes in the tree, X is the set of attributes, and $\mathcal{C}$ is the set of classes. Let $leaves(T)$ denote the set of all leaf nodes in T. A mapping d interprets a BDT if and only if $d(t) \in \mathcal{C}$ when $t \in leaves(T)$, and $d(t) \in X$ otherwise.

To assign a class to a letter $a \in \Sigma$, one needs to follow a path in T. This path is a sequence of nodes, which starts at the root of the tree and traverses it until reaching a leaf node. Formally, a path in T as a sequence $path_T(a) : t_0, \ldots, t_m$, such that $t_0 = \varepsilon$, $t_j \in \{t_{j-1}0, t_{j-1}1\}$ for all $j = 1, \ldots, m$, and $t_m \in leaves(T)$. The path is computed as follows: start at the root; when at inner node t_j, which is mapped to the variable x_i, make a test on a; proceed to the left child node $t_j 0$ if $x_i = 0$, proceed to the right child node $t_j 1$, otherwise; repeat until reaching a leaf

[3] A *pseudo-boolean function* is the analogous of a Boolean function, with the difference that the range of the function is any set other than $\mathbb{B}$.

[4] A binary tree T is a prefix closed set of positions $\mathcal{P}os(T) \subseteq \{0, 1\}^*$ with ε representing its root, such that (i) for every inner node $t \in \mathcal{P}os(T)$, nodes $t0$ and $t1$ are the left and right *child nodes*, respectively, and (ii) a node t is a leaf node when $t0 \notin \mathcal{P}os(T)$ and $t1 \notin \mathcal{P}os(T)$.

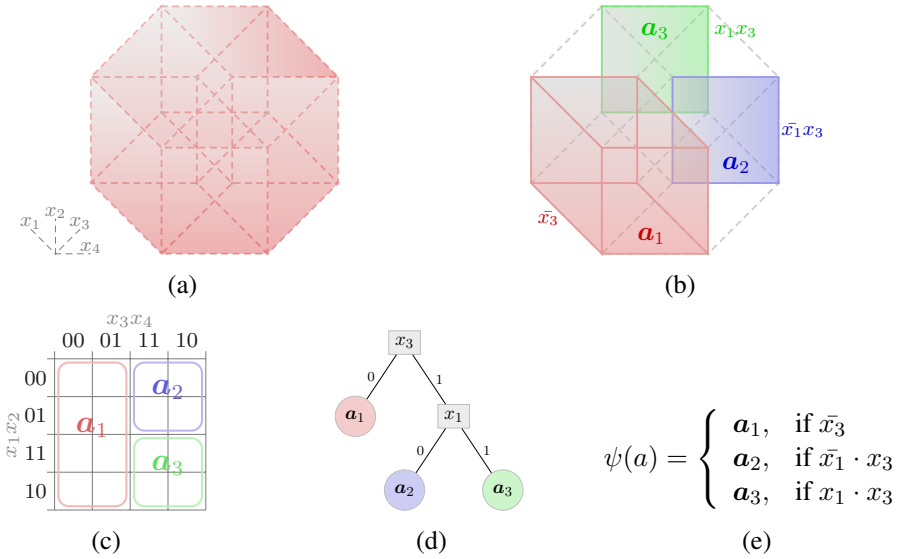


Figure 2.3: The (a) hypercube of four dimensions $\mathbb{B}^4$, and (b) a partition of it into three sub-cubes $\{\bar{x}_3, \bar{x}_1x_3, x_1x_3\}$, each one assigned to one class from $\mathcal{C} = \{\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3\}$. A representation of the classification function can be given as (c) a Karnaugh map, (d) a BDT, or (e) a pseudo-boolean function.

node. For each letter $a \in \Sigma$, there exists a unique path in T. We use $T(a)$ to denote the label of the leaf node reached by $path_T(a)$.

Each node $t \in \mathcal{P}os(T)$ is naturally associated with a sub-cube $[\![t]\!]$ of Σ, which consists of all letters $a \in \Sigma$ whose path in T contains the node t. Consequently, for every BDT T there exists an equivalent PBNF ψ that represents the same classification function. To find this PBNF, we work as follows. Let ϕ_t denote the cube associated to node t. This can be defined recursively as $\phi_\varepsilon = 1$ and $\phi_{t \cdot a} = \phi_t \cdot x_i^\kappa$, where x_i is the variable carried by node t and x^κ denotes x and $\bar{x}$ when $\kappa = 1$ and $\kappa = 0$, respectively. Let T a BDT, we define its equivalent PBNF as

$$\psi = \sum_{t \in leaves(T)} d(t) \cdot \phi_t$$

where $d(t)$ is the label associated to the leaf node t. To emphasize the relation between a PBNF ψ and its equivalent BDT we denote the latter by T_ψ.

A *Karnaugh map* is a classical representation of Boolean functions that offers a better visualization of functions involving a small number of variables (up to 5 or 6). A Karnaugh map is given in a matrix structure, where rows and columns are indexed by the values of some variables, and each cell contains the value of the function in the corresponding Boolean vector. For instance, in Figure 2.3c, rows and columns are indexed by the values of the pairs of variables (x_1, x_2) and

(x_3, x_4), respectively, and the cell in the second row and third column of the map contains the value (label) $\boldsymbol{a}_2$ since $\psi(0111) = \boldsymbol{a}_2$.

Example 2.2. Let the alphabet be the Boolean cube $\Sigma = \mathbb{B}^4$, see Figure 2.3a. A possible partition of this cube into three classes $\mathcal{C} = \{\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3\}$ is shown in Figure 2.3b. This partition can be equivalently represented by either the BDT in Figure 2.3d, or the Karnaugh map in Figure 2.3c, or the PBNF $\psi = \boldsymbol{a}_1 \cdot \bar{x_3} + \boldsymbol{a}_2 \cdot \bar{x_1} \cdot x_3 + \boldsymbol{a}_3 \cdot x_1 \cdot x_3$. Using the classification tree representation, the letter $a = (0, 0, 1, 0)$ follows the path $path_T(a) = \varepsilon, 1, 10$ in the tree, and thus it is assigned to the class $\boldsymbol{a}_2$, i.e., $\psi(a) = \boldsymbol{a}_2$. □

2.3 Learning Partitions

In the previous section, we have defined classification functions and have shown how a domain, an alphabet Σ, can be partitioned into classes. Moreover, we have presented various classification functions $\psi : \Sigma \to \mathcal{C}$, where $\mathcal{C}$ denotes a set of classes, typically finite and small.

However, a typical problem is to find such a representation, a classification function, when only a small set of classified elements from Σ is given. This problem is commonly encountered in machine learning and belongs in the context of *predictive* or *supervised learning*. To formalize the problem, let $S = \{(a^1, y^1), \ldots, (a^r, y^r)\} \subseteq \Sigma \times \mathcal{C}$ be a *labeled sample*. We assume the existence of an underlying classification function ψ, which is unknown outside S, such that $y^i = \psi(a^i)$ for all $i = 1, \ldots, r$. Our aim is to find a function ψ' which approximates well ψ on the labeled data from S, and, moreover, it generalizes well to make predictions on novel unseen inputs from Σ.

Depending on the set of labels $\mathcal{C}$, the problem is known as *classification* or *pattern recognition* when $\mathcal{C}$ is finite, or as *regression* when $\mathcal{C}$ is the set of reals. In this thesis, we restrict ourselves to a finite number of classes, and in this section, we describe classification learning techniques that are commonly used to learn the partitions on Boolean cubes presented in Section 2.2.

2.3.1 Learning Binary Decision Trees

Let S be a labeled sample from an unknown classification function ψ. In this section, we discuss the problem of learning a BDT to represent ψ that is compatible with the given sample S. As many such trees can be found, we prefer an optimal tree. Optimality usually refers to the tree size, such as its depth; the node purity, like the number of mismatching points in the sample; or other criteria such as

Ockham's razor. Since finding a globally optimal tree is NP-hard [BFSO84], a greedy splitting algorithm is commonly used to build the tree.

A greedy splitting algorithm works roughly as follows. It starts with a tree that consists of a single root node; all sample points are associated with this node. A node is said to be *pure* if all its sample points have the same label. For each impure node, two descendants are created and the sample is split among them based on the value of some selected variable x_i. The variable is chosen according to some purity measure, such as information gain, that characterizes the quality of the split based on each variable. The selection is greedy and does not depend on the optimality of following splits. The algorithm terminates when the tree becomes sample compatible and sends each sample point to a pure leaf node.

In the following, we describe more formally this learning algorithm. First, we present the case where the labeled sample is assumed to be given in advance. This algorithmic scheme is used by the most popular implementations for learning BDTs such as CART [BFSO84], ID3 [Qui86] and C4.5 [Qui93]. Then, we discuss the problem of learning a BDT when the sample is not completely provided before hand, but is updated during the learning process by adding new sample points. Popular incremental algorithms of this kind are, among others, the ID4 [SF86], ID5 [E88], and ID5R [Utg89] algorithms.

Using a Fixed Labeled Sample

Let $S = \{(a^1, y^1), \ldots, (a^r, y^r)\}$ be a labeled sample, where $y^i = \psi(a^i)$ and $\psi : \Sigma \to \mathcal{C}$ is an unknown classification function. We want to learn a BDT T such that $T \simeq T_\psi$. Each node $t \in \mathcal{P}os(T)$ associates to a set $S_{|t} = \{(x, y) \in S : t \in path(x)\}$ that consists of all sample points reaching node t. Note that $S_{|\varepsilon} = S$ for any labeled sample $S \subseteq \Sigma \times \mathcal{C}$. A node $t \in leaves(T)$ is said to be *consistent* with S whenever $T(x) = y$ for all $(x, y) \in S_{|t}$. A BDT is consistent with a labeled sample S if all its leaf nodes are consistent with S. A node $t \in \mathcal{P}os(T)$ is *pure* when all points in $S_{|t}$ carry the same label, i.e., $|\{y : (x, y) \in S_{|t}\}| = 1$.

Algorithm 1 Grow Tree

1: **initialize** (T, d), such that $\mathcal{P}os(T) = \{\varepsilon\}$ and $d(\varepsilon) = label(S)$
2: **while** $\exists\, t \in leaves(T)$ such that t is not pure **do**
3: Let $x_i = best\_attribute(S_{|t}, X)$
4: $\mathcal{P}os(T) = \mathcal{P}os(T) \cup \{t0, t1\}$
5: $d(t) = x_i$; $d(t0) = label(S_{|t0})$; $d(t1) = label(S_{|t1})$
6: **end while**

The greedy splitting algorithm is shown in Algorithm 1. The algorithm uses a quality measure to do splitting; the most common quality measures are presented

in detail below. We denote by $label(S)$ the most common label encountered in a sample S, that is, $label(S) = \arg\max_{c \in \mathcal{C}} |\{(x, y) \in S : y = c\}|$; we let $label(S)$ be equal to a random label from $\mathcal{C}$ when S is empty. The $best\_attribute(S, X)$ denotes the Boolean variable that performs the best split of a sample S into two subsets. The algorithm performs as follows. After being initialized to a one node tree, the BDT T is extended by splitting its leaf nodes until it becomes consistent with the sample. Whenever a leaf node t in T is not pure it becomes an inner node and two child leaf nodes $t0$ and $t1$ are added to $\mathcal{P}os(T)$ (line 4). The inner node t updates and maps to the attribute $x_i \in X$ that performs the best split according to the chosen quality measure (line 3); note that this variable does not appear so far in the path reaching the node t. The two new leaf nodes map to the most common label encountered in the sub-sample associated to them (line 5).

Splitting Quality Measures

To measure the spitting quality of an attribute and decide whether a split is better than another, we need to compare the gain achieved when splitting the sample S using each attribute. We compare on the gain on the node impurity and for this we use impurity measures. Other measures, as for instance on the size of the tree, are not considered here. As a further reference, one can consult [BFSO84, Mur12].

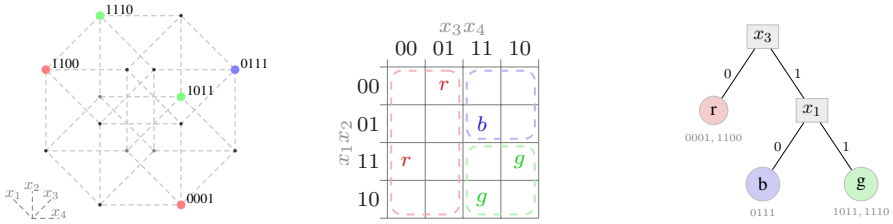
The set $\Delta^k \subseteq \mathbb{R}^k$, known as the standard k-simplex, is the set of all real tuples $(t_1, \cdots, t_k) \in \Delta^k$ such that $\sum_{i=1}^k t_i = 1$ and $t_i \geq 0$ for all i. A function $\mathcal{I} : \Delta^k \to \mathbb{R}$ is an *impurity measure* if i) it is symmetric with regard to its arguments, ii) it has minima at the vertices of Δ^k, and iii) it has one maximum at $(\frac{1}{k}, \ldots, \frac{1}{k})$. One can measure the *impurity of a labeled sample* S, with labels chosen from $\mathcal{C} = \{c_1, \cdots, c_k\}$, by letting

$$\mathcal{I}(S) = \mathcal{I}(p_1, \ldots, p_k) \text{ and } p_i = \frac{|\{(x, y) \in S \ : \ y = c_i\}|}{|S|}.$$

The impurity measures, most widely used in learning BDT, are the *misclassification rate*, the *entropy measure*, and the *Gini index*. These are defined on a tuple $(p_1, \ldots, p_k)$ as:

Misclassification rate:	$\mathcal{I}_{error}(p_1, \ldots, p_k) = 1 - \max_i p_i$
Entropy measure:	$\mathcal{H}(p_1, \ldots, p_k) = -\sum_{i=1}^k p_i \log_2 p_i$
Gini index:	$\mathcal{I}_G(p_1, \ldots, p_k) = 1 - \sum_{i=1}^k p_i^2$

Now we can measure the impurity gain when S is split using a specific attribute, and see whether a split is better than another. Let $x_i \in X$ be an attribute that splits S into two sub-samples $S_{x_i=0}$ and $S_{x_i=1}$, each containing the sample points that satisfy $x_i = 0$ and $x_i = 1$, respectively. The impurity gain achieved by splitting



(a) (b) (c)

Figure 2.4: The hypercube and Karnaugh map of $\mathbb{B}^4$ with the sample points indicated by their labels red (r), green (g), and blue (b). On the right is the BDT learned from S using Algorithm 1.

using attribute x_i is given by

$$Gain_{\mathcal{I}}(S, x_i) = \mathcal{I}(S) - \sum_{j=0,1} \frac{|S_{x_i=j}|}{|S|} \cdot \mathcal{I}(S_{x_i=j}),$$

where $\mathcal{I}$ is the impurity measure used. Let us demonstrate the learning algorithm with a simple example.

Example 2.3. Let the alphabet Σ be the Boolean hypercube $\mathbb{B}^4$ and let $\mathcal{C} = \{r, g, b\}$ be the set of labels, which correspond to the color names red (r), green (g), and blue (b), respectively. We use the learning algorithm described above to build a BDT. We let the labeled sample be $S = \{(0001, r), (0111, b), (1011, g), (1100, r), (1110, g)\}$. Figure 2.4a shows a 4-dimensional cube whose vertices denote the Boolean vectors and the sample points are denoted by label colors. The Karnaugh map in Figure 2.4b provides a more intuitive representation. As a split quality measure we use information gain (entropy measure). All the calculations needed for this example are shown in Table 2.1.

First, we initialize the BDT to (T, d) such that $\mathcal{P}os(T) = \{\varepsilon\}$ and $d(\varepsilon) = label(S) = r$. The node ε is not pure, i.e., $\mathcal{H}(S_{|\varepsilon}) > 0$. For this reason, the root node needs to be split. Comparing all possible splits, we find that the maximum information gain is achieved by splitting over x_3. Hence, the tree updates to $\mathcal{P}os(T) = \{\varepsilon, 0, 1\}$, $d(\varepsilon) = x_3$, $d(0) = label(S_{|0}) = r$, and $d(1) = label(S_{|1}) = g$. The leaf node $t = 0$ is pure, i.e., has entropy 0, and does not need to split further. The leaf node $t = 1$, on the other hand, is not pure. The best split is given for x_1, and the tree updates to $\mathcal{P}os(T) = \{\varepsilon, 0, 1, 00, 01\}$, $d(1) = x_1$, $d(00) = b$, and $d(01) = g$. Now, all leaf nodes are pure and the algorithm terminates. The final tree can be seen in Figure 2.4c. □

S	(r,g,b)	$\mathcal{H}(S)$	$Gain_{\mathcal{H}}(S,x_1)$	$Gain_{\mathcal{H}}(S,x_2)$	$Gain_{\mathcal{H}}(S,x_3)$	$Gain_{\mathcal{H}}(S,x_4)$
$S_{\|\varepsilon}$	$(2,2,1)$	1.52193	0.570951	0.170951	0.970951	0.170951
$S_{\|0}$	$(2,0,0)$	0	-	-	-	-
$S_{\|1}$	$(0,2,1)$	0.918296	0.918296	0.251629	0	0.251629
$S_{\|00}$	$(0,0,1)$	0	-	-	-	-
$S_{\|01}$	$(0,2,0)$	0	-	-	-	-

Table 2.1: Information gain for all samples and sub-samples of Example 2.3

Incremental Learning Algorithm

In this section, we change the setting of the learning problem and let the labeled sample be a stream of sample points received during the learning process. That is, the tree is learned and made compatible with the current sample when another new sample point arrives. The new observation incorporates to the sample and the tree updates according to the new information that is now available.

Formally, let $S^i = \{(a^1, y^1), \ldots, (a^i, y^i)\}$ denote the sample that consists of i sample points, and let (T^i, d^i) be a BDT tree compatible to S^i. When a new observation (a^{i+1}, y^{i+1}) arrives, we want to update T^i according to this new information. Several algorithms have been proposed in the literature to handle this problem. In the following, we explain some of these algorithms and discuss on their efficiency and complexity results.

One naive algorithm, which also appears in [SF86] for comparison purposes, uses Algorithm 1 as an incremental algorithm. Whenever a new observation or a set of new observations arrives, this is added to the sample. Then, the Algorithm 1 is called and outputs a new tree which is constructed from scratch based on the updated sample.

It is observed that a large part of the tree need not change at all, making the previous method unnecessarily costly. In [SF86], the authors propose an incremental algorithm where only a sub-tree needs to be rebuilt. The existing tree T^i is revised based on the new sample point (a^{i+1}, y^{i+1}), which traverses the nodes of the tree starting from the root. The quality measures are updated including the new sample point. If for a node $t \in \mathcal{P}os(T^i)$ the selected attribute is not the best anymore, then the sub-tree rooted at t is deleted and rebuilt. The sub-tree with root t is reconstructed using Algorithm 1 on the sample $S^i_{|t} \cup \{(a^{i+1}, y^{i+1})\}$.

To avoid preserving the whole set of sample points that have been seen so far, the ID4 algorithm [SF86] uses a series of tables, located at each node, that carry all information needed to calculate the quality measures when revising the tree.

Finally, another algorithm that is proposed is the ID5 algorithm [E88], which is followed by two improved versions, the ID5R [Utg89] and ITI [UBC97] algorithms. These algorithms use an alternative revising mechanism. That is, whenever

a sub-tree of T^i is found such that the splitting attribute at its root is not the best split, then this is only restructured instead of reconstructed as in ID4. To restructure the sub-tree, the algorithm uses a *pull-up* method in a way that all attributes at the nodes are the same.

A worst-case analysis and comparison of these incremental algorithms can be found in [Utg89]. It is shown that the ID4 algorithm cannot incrementally learn the parity function when the sample is given in a certain order, because it has always to update the whole tree. The cost of calculating the gain function and choosing the best attribute is significantly lower in the case of ID5R.

In this section, we restricted our discussion to algorithms where a) the order in which sample points arrive does not impact the final result, and b) the final BDT, which is constructed incrementally, is the same as if the tree is built using the non-incremental algorithm discussed in Section 2.3.1.

In the literature, one can find many extensions and variations of the algorithms discussed above that handle the same or slightly different settings of the learning BDT problem. For instance, when the sample is noisy or incomplete, most algorithms include a pruning phase after the tree has been fully constructed in order to avoid overfitting of the data [BFSO84, Qui93]. During this phase the stopping criteria are relaxed and some branches of the tree are pruned back whenever they do not add any significant gain to the final tree.

CHAPTER 3
Language Identification

This chapter introduces in more detail the problem of learning formal languages, that is a subset of Σ^*, also known as language identification. We start, in Section 3.1, by connecting language identification to the more general machine learning context. Then, in Section 3.2, we provide the state of the art in the field and explain the notions of identification in the limit and query learning. The presentation of the active learning algorithm L^* is given in Sections 3.3, and it is followed by a description of some well-known variations. A full reference on the topics presented in this chapter can be found in [Ang87, BR05, DlH10].

3.1 Connection with Machine Learning

Machine learning has gained popularity in the last decade with a variety of applications being associated with the field. Recognizing patterns after having observed a relatively small amount of data is the main task of almost all machine learning techniques.

A general and abstract description of a problem in this context is the following. Let $f : \Sigma \to \mathcal{C}$ be a function, where Σ is a set of objects and $\mathcal{C}$ is a set of labels. Given a labeled sample S that consists of pairs $(a, f(a)) \in \Sigma \times \mathcal{C}$, construct a function f' that exactly or approximately agrees with f on S, and behaves well on unseen data. Depending on whether the labels are taken from a finite or infinite set $\mathcal{C}$, the problem is known as *classification* or *regression* problem, respectively.

Machine learning problems, in the setting described above, are called *supervised*. In an *unsupervised* setting, the values of $\mathcal{C}$ are not provided, and the objects from S are grouped, clustered and classified, according to similarity measures.

Another important factor in learning problems is whether the learning is done *online* or *offline*. In online learning, the sample S gets updated during the learning

process and the classification or regression function can be updated to accommodate the new information that arrives. On the other hand, in offline problems the learning process starts after all sample data is given.

The learning algorithm can have an *active* role in the process by choosing the sample points that are believed to provide more information. In such a case the algorithm is called *active*. The algorithm is called *passive* otherwise.

Even though the primary goal of machine learning is to learn a *model* (or *function*) and then use it for prediction, a conjectured model can be useful also in other contexts. For instance, data may come from a system that needs to be analyzed or checked for correctness. Such a system might be a complex program or a cyber-physical system with multiple components and embedded sub-systems. Often, a formal model of such a system is not available or particular components of the system are not accessible. As a result, the system can be viewed as a black box, or sometimes a gray box, where learning is used to identify the underlying formal model.

One main characteristic that differentiates learning models as system representations from conventional machine learning problems is that typically the inputs are *sequences* of events in contrast to the *static* objects that are more commonly used in machine learning. Typically, instances stand for behaviors of some dynamical system, a continuous one or an automaton.

Formally, the problem of learning an unknown system or model from observations is known as *inductive* or *model inference* or, more generally, *system identification*. To characterize the type of the inductive inference problem, one needs to define three main aspects of the problem. First, the type of model that is to be learned; second, the way the observations are given; and third, the rules used to measure success. For instance, data may be given in an on-line fashion or in a batch; data may contain observations that are labeled or not; the underlying model is assumed to be of a certain type.

When the target model is assumed to be a language, the problem of inductive inference is called *language identification*. In this case, the learned model is usually a grammar or an automaton. Language identification can be found in the literature under the names of *grammatical inference*, *grammar induction*, *automata learning*, or *automata inference*. Language identification is concerned with learning language representations from information, which can be text, examples and counter-examples, or anything that can provide an insight into the elements of the languages being acquired.

Although, the terms induction and inference are interchangeably used, their meaning is not identical [DlH10]. In particular, when one has data and tries to find a representation that better explains them, e.g., a grammar or an automaton, the term *induction* is used; the derived model is basically used for prediction. On

the other hand, the term *inference* refers to problems where we assume that there exists an underlying target language, from which the data emerge, and one tries to discover it; the learning process, which is being examined and measured, is more important in this case.

Language identification is considered a sub-field that lies at the intersection of many fields such as: artificial intelligence, algorithmic learning theory, machine learning, statistics, formal languages and automata theory.

3.2 Learning Languages

The problem of learning finite state systems was first suggested by Moore [Moo56], who described the situation where you find a black box with buttons and lights and try to infer the internal structure (identification) and solve other problems based on pushing the buttons and observing the lights. One of the first positive results on automata was given by Gold in the context of inductive inference, in an on-line and passive setting, where one observes a sequence of classified words and at each point in time maintains an automaton compatible with the sample seen so far. He showed that an algorithm based on the enumeration of all finite automata, will identify regular languages in the limit [Gol67, OG92]. This means that given any presentation of any language in the class the learner will produce only a finite number of wrong representations, and therefore converge on the correct representation in a finite number of steps, without however necessarily being able to announce its correctness since a counter-example to that representation could appear as an element arbitrarily long after.

Most algorithms for learning automata from examples, starting with the seminal work of Gold [Gol72] and culminating in the well-known L^* algorithm of Angluin [Ang87], are based on the concept of Nerode right-congruence relation [Ner58], which declares two *input histories* as equivalent if they lead to the same *future continuations*. In passive learning problems, given a sample of labeled examples, learning the smallest deterministic finite state automaton representation of the target regular language is NP-complete [Gol78]. Additionally, it cannot be approximated within any polynomial [PW93].

We are interested in active learning algorithms where the learner can interact with the teacher and select the words in the membership queries. The L^* algorithm [Ang87], on which this thesis is based, is one such algorithm. It overcomes the negative complexity results by introducing a minimally-adequate teacher, who can answer also equivalence queries that either confirm a conjectured automaton or provide a counter-example. In this setting, regular languages have been shown to be learnable using a polynomial number of queries.

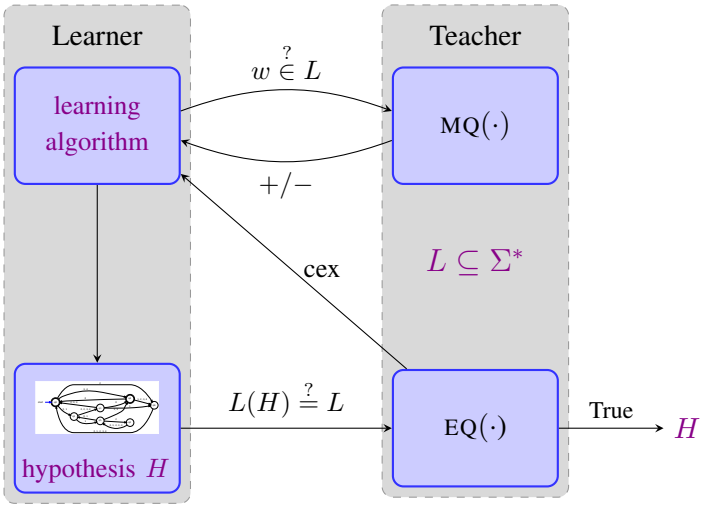


Figure 3.1: The Teacher-Learner model consists of a learner (the learning algorithm) and a teacher. The teacher can answer two types of queries about the target language, membership queries (MQ), and equivalence queries (EQ). The learner asks MQ's until being able to construct a hypothesis H. Then, the validity of an hypothesis is checked by an EQ. If H is correct the algorithm terminates returning H, otherwise, a counter-example (cex) is returned and the learner improves the hypothesis.

3.3 The L^* Learning Algorithm

Represented by the *learner*, the learning algorithm is designed to infer an unknown regular language L, the *target language*. The learner aims to construct a deterministic finite state automaton that recognizes the target language by gathering information from a *teacher*. The *teacher*, who knows the target language, can provide information about it by answering queries.

The algorithmic scheme used in L^*, is shown in Figure 3.1. The learner starts by asking *membership queries*, i.e., whether a word belongs to the target language or not. All information provided is suitably gathered in a table structure, the *observation table*. When the information is sufficient, the learner constructs a *hypothesis automaton* and poses an *equivalence query* to the teacher. If the answer is positive, i.e., the hypothesis suggested by the learner is the right one, the algorithm terminates and returns the conjectured automaton. Otherwise, the teacher responds to the equivalence query with a *counter-example*, a word misclassified by the conjectured automaton. Then, the learner incorporates the information provided by the counter-example into the table and repeats the procedure until a correct hypothesis is constructed.

3.3.1 Observation table

An active learner organizes all information about the target language in a table structure, called the *observation table*. To construct a deterministic automaton, the learner needs to determine a set of states, with a subset of those determined as accepting, and a transition function. Hence, the observation table consists of three main parts: a set of candidate states and transitions, that is, a set of prefixes $S \cup R \subseteq \Sigma^*$ that label the rows of the table; a set of distinguishing suffixes $E \subseteq \Sigma^*$ that label the columns of the table and help to distinguish between states; and, the main body of the table, which consists of the classification of sequences whose membership is known. The rows in the table induce a Nerode right-congruence which is used to construct the hypothesis automaton.

The elements of S admit a tree structure isomorphic to a *spanning tree* of the transition graph, rooted in the initial state. Each $s \in S$ corresponds to a state q of the automaton for which s is an *access sequence*, that is, one of the shortest words that lead from the initial state to q. The elements of R hold information about the back- and cross-edges in the automaton.

Formally, let Σ be a finite alphabet. A prefix-closed set $S \uplus R \subset \Sigma^*$ is a finite *balanced* Σ*-tree* if for all letters $a \in \Sigma$: 1) for every $s \in S$, $s \cdot a \in S \cup R$, and 2) for every $r \in R$, $r \cdot a \notin S \cup R$. Elements of R are called *boundary elements* (or *leaves*).

Definition 3.1 (Observation Table)**.** An *observation table* is a tuple $T = (\Sigma, S, R, E, f)$ such that Σ is a finite alphabet; $S \uplus R$ is a finite balanced Σ-tree, with R being its boundary; E is a suffix-closed subset of Σ^*; and $f : (S \cup R) \cdot E \to \{-, +\}$ is a classification function.

Let $L \subseteq \Sigma^*$ be the target language. The classification function f of the table is a restriction of the characteristic function of L. The set $M = (S \cup R) \cdot E$ consists the *sample* associated to the table. For all $s \cdot e \in M$, the element $f(s \cdot e)$ is placed in the (s, e) position of the table. We say that a table has no holes when membership is known for all words in M.

With every prefix $s \in S \cup R$, we associate the residual $f_s : E \to \{-, +\}$, defined as $f_s(e) = f(s \cdot e)$, which characterizes the row of the observation table labeled by s. Two sequences s and r are considered as Nerode equivalent with respect to L, i.e, $s \sim_L r$, if $f_s = f_r$. Based on this right congruence relation, the learner can build an automaton from the table which, moreover, is minimal. However, the table should satisfy certain conditions.

Definition 3.2. An observation table $T = (\Sigma, S, R, E, f)$ is:

– *closed* if for every $r \in R$, there exists an $s \in S$, such that $f_r = f_s$;

– *reduced* if for every $s, s' \in S$, $f_s \neq f_{s'}$;

– *consistent* if for every $s, s' \in S$, $f_s = f_{s'}$ implies $f_{s \cdot a} = f_{s' \cdot a}, \forall a \in \Sigma$.

Theorem 3.3. *From a closed and reduced table, one can construct a finite deterministic automaton that is compatible with the sample.*

Proof. Let $T = (\Sigma, S, R, E, f)$ be a closed and reduced observation table. For a closed and reduced table we let $g : R \to S$ be a function that maps every $r \in R$ to the unique $s \in S$ such that $f_s = f_r$.

We can define the automaton $\mathcal{A}_T = (\Sigma, Q, q_0, \delta, F)$ such that $Q = S$, $q_0 = \varepsilon$, $F = \{s \in S : f_s(\varepsilon) = +\}$ and

$$\delta(s, a) = \begin{cases} s \cdot a, & \text{when } s \cdot a \in S \\ g(s \cdot a), & \text{when } s \cdot a \in R \end{cases}$$

□

Note that a reduced table is trivially consistent. Moreover, every consistent non reduced table has a reduced form.

To explain this better, a table is not reduced when there exist s_1 and $s_2 \in S$ such that $f_{s_1} = f_{s_2}$. To reduce the table, we select one out of the two prefixes, preferably the longest one, and move it from S to R. This step may repeat several times until the table becomes reduced and S consists of rows that distinct. Note that, after converting a non reduced table to a reduced one, the set R is not necessarily equal to $S \cdot \Sigma$, and it may contain redundant elements.

To construct a finite deterministic automaton of a closed and consistent observation table that is compatible with the sample, we let the set of states of the automaton, which is a subset of S, to be the set of all the prefixes $s \in S$ with distinguishable residuals f_s. The transition function is constructed similarly as for a reduced table.

3.3.2 The Learning Algorithm

The learning algorithm, shown in Algorithm 2, proceeds as follows. The learner initializes the table to $T = (\Sigma, S, R, E, f)$ such that $S = E = \{\varepsilon\}$ and $R = \Sigma$. The table is then filled in, i.e., the classification f is determined for all words in the sample M that are missing, by asking membership queries to the teacher.

The learner attempts to keep the table closed and consistent at all times. The table T is not closed when there exists a prefix $r \in R$ such that f_r is different from f_s for all $s \in S$. To close the table, the learner moves r from R to S and adds to R the Σ-successors of r, i.e., all words $r \cdot a$ for $a \in \Sigma$, see Procedure 3. The table T

Algorithm 2 Main Learning Algorithm

```
1: learned = false
2: Let T = (Σ, S, R, E, f) such that S = E = {ε} and R = Σ   ▷ initialize table
3: Ask MQ's to fill in T
4: repeat
5:     while T is not closed or not consistent do
6:         CLOSE
7:         CONSISTENT
8:     end while
9:     Construct A_T
10:    if EQ(A_T) then                                   ▷ check hypothesis
11:        learned = true
12:    else                                  ▷ a counter-example w is provided
13:        COUNTEREX(A_T, w)                          ▷ process counter-example
14:    end if
15: until learned
16: return A_T
```

is inconsistent when there exist $s_1, s_2 \in S$, $e \in E$, and $a \in \Sigma$ such that $f_{s_1} = f_{s_2}$ but $f_{s_1}(a \cdot e) \neq f_{s_2}(a \cdot e)$. To fix the inconsistency, the string $a \cdot e$ is added to E as a new distinguishing experiment, see Procedure 4. In both cases the extended table is filled in by asking membership queries.

When the observation table $T = (\Sigma, S, R, E, f)$ becomes both closed and consistent, following Theorem 3.3, the learner makes a hypothesis automaton $\mathcal{A}_T = (\Sigma, Q, \varepsilon, \delta, F)$, where $Q \subseteq S$ is the set of states, $F \subseteq Q$ is the set of final states, and δ is the transition function. The set of states consists of all elements $s \in S$ with distinct residual, i.e., $s, s' \in Q$ if and only if $f_s \neq f_{s'}$, and moreover, for all $s \in S$ there exists $s' \in Q$ such that $f_s = f_{s'}$. When the table is reduced, then $Q = S$. A state $s \in Q$ is final when $f_s(\varepsilon) = +$. The transition function is defined as $\delta(s, a) = s'$ such that $f_{s'} = f_{s \cdot a}$, for all $s \in Q$ and $a \in \Sigma$.

When a counter-example is presented, the learner incorporates this into the sample and tries to construct another hypothesis. Variants of the L^* algorithm differ in the way they treat counter-examples. Subsequently, we present how a counter-example is treated in the algorithm L^*, followed by two variations, proposed in [MP95] and [RS93], respectively.

In any case, the table is extended and new states are added to the hypothesis. The learner asks new membership queries to fill in the table, closes the table and makes it consistent when needed, before making the new conjecture. This repeats until the teacher responds positively to an equivalence query.

Procedure 3 Table Closing

procedure CLOSE
  Given $r \in R$ such that $\forall s \in S$, $f_r \neq f_s$
  $S = S \cup \{r\}$
  $R = (R - \{r\}) \cup r \cdot \Sigma$
  Ask MQ's to fill in T
end procedure

Procedure 4 Make Table Consistent

procedure CONSISTENT
  Given $s, s' \in S$, $a \in \Sigma$ and $e \in E$ such that
    $f_s = f_{s'}$ and $f_{s \cdot a}(e) \neq f_{s' \cdot a}(e)$
  $E = E \cup \{a \cdot e\}$
  Ask MQ's to fill in T
end procedure

Counter-Example Treatment

The classic L^* algorithm. The original algorithm [Ang87] adds all the *prefixes* of a counter-example w to S, see Procedure 5. Then all continuations of the new prefixes are added to R, such that $S \cdot \Sigma \subseteq S \cup R$ holds. The learner fills in the table by asking membership queries. The counter-example treatment creates an inconsistency or renders the table non closed.

Notice that this treatment of the counter-example adds many redundant rows to the table and only a subset of S forms the set of states in the conjectured automaton.

Procedure 5 Counter-Example Treatment: L^*

procedure COUNTEREX
  let $w \in \Sigma^*$ be a counter-example
  for $u \in \Sigma^*$ such that $w = u \cdot v$, $v \in \Sigma^*$ **do**
    $S = S \cup \{u\}$ ▷ add w and all its prefixes to S
    $R = R \cup u \cdot \Sigma$
  end for
  Ask MQ's to fill in T
end procedure

Adding Suffixes. This version is proposed in [MP95] and was initially used for learning ω-regular languages. The learner adds all the *suffixes* of a counter-example

w to E, see Procedure 6. Then the learner fills in the table.

The advantage of this approach is that the table always remains reduced and thus consistent, with S corresponding exactly to the set of states. A disadvantage, however, is the possible introduction of redundant columns that do not contribute to further discrimination between states.

Breakpoint. Another variation of the counter-example treatment in this algorithm is due to [RS93] and is referred in [BR05] as the *reduced algorithm*. In this variation, the counter-example is first processed by the so-called *breakpoint* method, and then only the suffixes necessary to detect a new state are added to E.

The breakpoint method, see Procedure 7, searches for a cut point in the counter-example, the *breakpoint*. This provides a factorization of the counter-example whose suffix detects a new state. Formally, let $w = a_1 \cdots a_{|w|}$ be a counter-example to a hypothesis $\mathcal{A}_T$ associated with a table T. An *i-factorization* of w is $w = u_i \cdot a_i \cdot v_i$ such that $u_i = a_1 \cdots a_{i-1}$ and $v_i = a_{i+1} \cdots a_{|w|}$. Let $s_i = \delta(\varepsilon, u_i \cdot a_i)$ be the state of $\mathcal{A}_T$ (element of S in table T) reached after reading $u_i \cdot a_i$ and let s_0 be the initial state.

Proposition 3.4 (Breakpoints [RS93])**.** *If w is a counter-example to $\mathcal{A}_T$ then there is an i-factorization of w such that $f(s_{i-1} \cdot a_i \cdot v_i) \neq f(s_i \cdot v_i)$.*

In other words, adding v_i to E distinguishes $s_{i-1} \cdot a_i$ from s_i, makes the table not closed and leads to the identification of $s_{i-1} \cdot a_i$ as a new state. The breakpoint method iterates over i values between 1 and $|w|$ to find new states and experiments that distinguish them from old states. Note that a counter-example may admit more than one breakpoint.

We let i take values in a monotonically decreasing order. The reason for doing this is to favor the detection of smaller distinguishing strings that are added as suffixes to the table. The iteration in an increasing order can be used as well, making the breakpoint method more effective when we need to add prefixes to the table. Alternatively, we can proceed by choosing the values i in a binary search manner and be sure to find a breakpoint in logarithmic time, or in an exponential search manner when of favor to short suffixes. These different iteration methods are experimentally tested in [IS14].

Example 3.5. We illustrate the behavior of the L^* algorithm while learning the target language $L = a \cdot \Sigma^*$ over $\Sigma = \{a, b\}$. We use $+w$ to indicate a counter-example $w \in L$ rejected by the conjectured automaton, and $-w$ otherwise.

Initially, the observation table is $T_0 = (\Sigma, S, R, E, f)$ with $S = E = \{\varepsilon\}$ and $R = \Sigma$. We ask membership queries for all words in $(S \cup R) \cdot E = \{\varepsilon, a, b\}$ to obtain table T_0, shown in Figure 3.2. The table is not closed so we move a to S, add its continuations, $a \cdot a$ and $a \cdot b$ to R, and ask membership queries to obtain the

Procedure 6 Counter-Example Treatment: Adding Suffixes

```
1: procedure CounterEx
2:     let w ∈ Σ* be a counter-example
3:     for each v ∈ Σ* such that w = u · v, u ∈ Σ* do
4:         E = E ∪ {v}                          ▷ add w and all its suffixes to E
5:     end for
6:     Ask MQ's to fill in T
7: end procedure
```

Procedure 7 Counter-Example Treatment: Breakpoint

```
1: procedure CounterEx
2:     let w ∈ Σ* be a counter-example
3:     for i = |w|, ..., 1 do
4:         let u_i · a_i · v_i be the i-factorization of w
5:         if f(s_{i-1} · a_i · v_i) ≠ f(s_i · v_i) then
6:             E = E ∪ {v_i}
7:             break
8:         end if
9:     end for
10:    Ask MQ's to fill in T
11: end procedure
```

T_0

	ε
ε	$-$
a	$+$
b	$-$

T_1

	ε
ε	$-$
a	$+$
b	$-$
$a \cdot a$	$+$
$a \cdot b$	$+$

T_2

	ε
ε	$-$
a	$+$
b	$-$
$b \cdot a$	$-$
$a \cdot a$	$+$
$a \cdot b$	$+$
$b \cdot b$	$-$
$b \cdot a \cdot a$	$-$
$b \cdot a \cdot b$	$-$

T_3

	ε	a
ε	$-$	$+$
a	$+$	$+$
b	$-$	$-$
$b \cdot a$	$-$	$-$
$a \cdot a$	$+$	$+$
$a \cdot b$	$+$	$+$
$b \cdot b$	$-$	$-$
$b \cdot a \cdot a$	$-$	$-$
$b \cdot a \cdot b$	$-$	$-$

Figure 3.2: Observation tables of Example 3.5 when learning $a \cdot \Sigma^*$ using the counter-example treatment method that is used in the classic L^* algorithm.

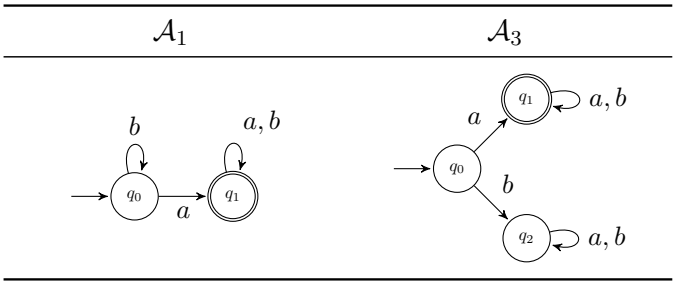


Figure 3.3: Hypothesis automata of Example 3.5 when learning $a \cdot \Sigma^*$.

closed table T_1. From T_1 we construct the hypothesis automaton $\mathcal{A}_1$ of Figure 3.3. In response to the equivalence query for $\mathcal{A}_1$, counter-example $-b \cdot a$ is presented.

First, let us use the classical counter-example treatment used by the L^* algorithm. In this case, the learner adds all prefixes of the counter-example $b \cdot a$ to the set of prefixes, that is, it adds the prefixes b and ba to S. All successors are added to R, resulting in table T_2 of Figure 3.2. This table is not consistent since the two elements ε and b in S are considered equivalent but their a-successors a and $b \cdot a$ are not. Adding a to E and asking membership queries yields a closed and consistent table T_3. The derived automaton $\mathcal{A}_3$ is the desired automaton that recognizes the target language L and the algorithm terminates.

In the following, we will use the same example, but we will apply another method for treating the counter-example $-b \cdot a$. We have chosen an example where the hypotheses remain the same, but this is frequently not the case. We focus here on the different observation tables T_2 and T_3 that are produced for each method.

Let us use the suffixes method for treating $-b \cdot a$, we add all suffixes, i.e., a and $b \cdot a$, to E. The resulting table is thus T_2^m, see Figure 3.4. We recall here, that with this method the learner need not check for consistency, as the table is always kept consistent. After closing the table T_2^m updates T_3^m. The hypothesis associated to this table is A_3, shown in Figure 3.3. The algorithm terminates as A_3 is correct.

Finally, we analyze the counter-example $-b \cdot a$ using the breakpoint method. This results in finding one distinguishing string that is enough to be added to the table and to distinguish some states. The i-factorization for $i = 1$ yields such a suffix, since $f(b \cdot a) \neq f(\varepsilon \cdot a)$, as Procedure 7 indicates. As a result, the suffix $v_1 = a$ is added to E and the new table is T_2^b is shown in Figure 3.5. After making it closed, the table updates to T_3^b and the hypothesis A_3, see Figure 3.3, is conjectured. The algorithm terminates with this hypothesis being correct.

T_2^m

	ε	a	$b \cdot a$
ε	$-$	$+$	$-$
a	$+$	$+$	$+$
b	$-$	$-$	$-$
$a \cdot a$	$+$	$+$	$+$
$a \cdot b$	$+$	$+$	$+$

T_3^m

	ε	a	$b \cdot a$
ε	$-$	$+$	$-$
a	$+$	$+$	$+$
b	$-$	$-$	$-$
$a \cdot a$	$+$	$+$	$+$
$a \cdot b$	$+$	$+$	$+$
$b \cdot a$	$-$	$-$	$-$
$b \cdot b$	$-$	$-$	$-$

Figure 3.4: Observation tables of Example 3.5 when learning $a \cdot \Sigma^*$ using the counterexample treatment of the suffixes method.

T_2^b

	ε	a
ε	$-$	$+$
a	$+$	$+$
b	$-$	$-$
$a \cdot a$	$+$	$+$
$a \cdot b$	$+$	$+$

T_3^b

	ε	a
ε	$-$	$+$
a	$+$	$+$
b	$-$	$-$
$a \cdot a$	$+$	$+$
$a \cdot b$	$+$	$+$
$b \cdot a$	$-$	$-$
$b \cdot b$	$-$	$-$

Figure 3.5: Observation tables of Example 3.5 while learning $a \cdot \Sigma^*$ and using the counterexample treatment of the breakpoint method.

4 CHAPTER
Symbolic Automata

4.1 State of the Art

In recent years, interest in languages defined over large and infinite alphabets has increased both for theoretical and practical consideration [Seg06]. When regular languages are defined over a large or infinite domain it is impractical or unfeasible to use standard finite automata to represent them. Several authors consider automata models that use infinite alphabets, namely, *finite memory automata*, also known as *register automata*, [KF94, NSV04]; *pebble automata* [NSV04]; *data automata* [BMS+06]; and *variable automata* [GKS10]. These are finite state automata that use a finite memory, e.g., registers, pebbles, or variables, to handle the infinite nature of the alphabet.

Other automata models over infinite alphabets are finite state automata, where transitions are labeled by predicates applicable to the alphabet in question. Hence, transitions are associated with formulas representing sets of letters rather than individual letters resulting in automata with finitely many transitions and thus with a finite representation. No memory is used and predicates are evaluated only on the current input. The idea of such automata was mentioned already in [Wat96]. However, it was formally introduced and studied later, in [VNG01], under the name of *predicate-augmented finite state recognizers*. In a similar flavor, *lattice automata* [LGJ07] use elements of an atomic lattice to label transitions rather than general predicates. *Interval automata*, which are automata where a move is realized whenever the input letter belongs to a predefined interval, can be viewed as a special case of both lattice automata and predicate-augmented finite state recognizers. Automata with predicates were further studied in [VDHT10, Vea13], under the name *symbolic automata* and predicates are chosen from a Boolean Algebra. Finally, one can find a special variant of symbolic automata in [HJJ+95, BKR96, YBCI08],

where transition relations are represented as Multi-terminal Binary Decision Diagrams.

Hereafter, we use the automata, as defined in [VNG01, Vea13], and refer to them as *symbolic automata*. The input, which comes from a concrete alphabet that may be large or infinite, is treated in a symbolic way. Transitions are labeled by symbols taken from a finite symbolic alphabet. Each symbol stands for a predicate and represents a non-empty set of concrete letters. Symbolic automata are shown to be closed under union, complement, concatenation, and star operations [VNG01, YBCI08]. Using the product construction one can show that they are also closed under intersection and difference [VNG01, VBDM10, HV11]. Moreover, symbolic automata are determinizable, they can be made complete, and ε-transitions can be eliminated [VNG01, VDHT10]. Minimization algorithms are adapted to the symbolic setting and studied in detail in [DV14]. Equivalence checking is decidable and counter-examples can be found in the case where two symbolic automata are not equivalent [VHL+12, DV14].

Their succinct representation makes symbolic automata suitable for language representation not only when the input alphabet is infinite, but also when the alphabet is finite but large, e.g., the Unicode alphabet UTF-16. The smaller size of the automaton allows faster implementation of various operations [VNG01]. The necessity of introducing and studying symbolic automata derives from several applications which have their roots in quite diverse areas such as natural language processing [VNG01]; support for regular expressions in the context of static and dynamic program analysis [VDHT10]; constraint solving and symbolic analysis with SMT solvers [VBDM10]; support for some specific regular expressions, e.g., LIKE-patterns, used in the context of symbolic analysis of database queries [VGDHT09, VTDH10]; and security analysis of string sanitizers [Vea13].

In this chapter, we first introduce symbolic automata (Section 4.2) and then discuss their closure properties and canonical form (Section 4.3). We end the chapter with some examples of symbolic automata defined over specific domains, such as numerical alphabets, e.g., $\mathbb{N}, \mathbb{R}$, or sets of vectors, e.g., $\mathbb{B}^n$.

4.2 Definition

Let Σ be a large, possibly infinite, alphabet, to which we will refer from now on as the *concrete* alphabet. A *symbolic automaton* over Σ is a finite automaton where each state has a small number of outgoing transitions labeled by *symbolic letters* (or *symbols*) taken from a finite alphabet $\mathbf{\Sigma}$; we call this the *symbolic alphabet*. Symbols represent subsets of Σ, which may differ from state to state.

Formally, let $\mathbf{\Sigma}$ be a disjoint union of finite symbolic alphabets of the form $\mathbf{\Sigma}_q$, each associated with a state q of the automaton. We choose a state-depended

alphabet as being more intuitive and efficient for the purposes of the current thesis. Nevertheless, for language-theoretic studies, one may find that a globally defined symbolic alphabet, where the alphabet is the same for all states, is a more suitable option. A state local alphabet can be easily transformed into a global one; the procedure is presented in detail at the end of this section.

Concrete letters are mapped to symbols through a family of mappings ψ decomposable into state-specific mappings $\psi_q : \Sigma \to \boldsymbol{\Sigma}_q$, where q is a state in the automaton. The Σ-*semantics* of a symbolic letter $\boldsymbol{a} \in \boldsymbol{\Sigma}$ denotes the subset of concrete letters from Σ that take the transition labeled by $\boldsymbol{a}$ in the automaton. This can be seen as a mapping $[\![\cdot]\!] : \boldsymbol{\Sigma} \to 2^{\Sigma}$ defined as $[\![\boldsymbol{a}]\!] = \{a \in \Sigma : \psi_q(a) = \boldsymbol{a}\}$ for any symbol $\boldsymbol{a} \in \boldsymbol{\Sigma}_q$. We often omit ψ and ψ_q from the notation and use $[\![\boldsymbol{a}]\!]$ when ψ, which is always present, is clear from the context.

The Σ-semantics is extended to symbolic words of the form $\boldsymbol{w} = \boldsymbol{a}_1 \cdots \boldsymbol{a}_{|\boldsymbol{w}|} \in \boldsymbol{\Sigma}^*$ as the concatenation of the concrete one-letter languages associated with the respective symbolic letters or, recursively speaking, by letting $[\![\boldsymbol{\varepsilon}]\!] = \{\varepsilon\}$ and $[\![\boldsymbol{w} \cdot \boldsymbol{a}]\!] = [\![\boldsymbol{w}]\!] \cdot [\![\boldsymbol{a}]\!]$ for $\boldsymbol{w} \in \boldsymbol{\Sigma}^*$, $\boldsymbol{a} \in \boldsymbol{\Sigma}$.

The symbolic automaton is *deterministic* when the family ψ consists of functions; and, it is *complete* when the mappings in ψ are total. Moreover, if the functions in ψ are surjective, there are no redundant symbols in $\boldsymbol{\Sigma}$, i.e., symbols whose semantics is empty. As a result, a total surjective function ψ_q, where q is a state in the symbolic automaton, implies that the set of Σ-semantics of all symbols in $\boldsymbol{\Sigma}_q$, $\{[\![\boldsymbol{a}]\!] : \boldsymbol{a} \in \boldsymbol{\Sigma}_q\}$, forms a *partition* of Σ. This should hold for all states in the automaton for it to be complete and deterministic. Formally,

Definition 4.1 (Symbolic Automaton)**.** A *deterministic symbolic automaton* is a tuple $\boldsymbol{\mathcal{A}} = (\Sigma, \boldsymbol{\Sigma}, \psi, Q, \boldsymbol{\delta}, q_0, F)$, where

- Σ is the input alphabet,
- $\boldsymbol{\Sigma}$ is a finite alphabet, decomposable into $\boldsymbol{\Sigma} = \biguplus_{q \in Q} \boldsymbol{\Sigma}_q$,
- $\psi = \{\psi_q : q \in Q\}$ is a family of total surjective functions $\psi_q : \Sigma \to \boldsymbol{\Sigma}_q$,
- Q is a finite set of states,
- q_0 is the initial state,
- F is the set of accepting states, and
- $\boldsymbol{\delta} : Q \times \boldsymbol{\Sigma} \to Q$ is a partial transition function decomposable into a family of total functions of the form $\boldsymbol{\delta}_q : \{q\} \times \boldsymbol{\Sigma}_q \to Q$.

Let $w = a_1 \cdots a_n \in \Sigma^*$ be a concrete word. A run of w on $\boldsymbol{\mathcal{A}}$ produces a path $P(w) : q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$ that corresponds to the symbolic path $\boldsymbol{P}(w) : q_0 \xrightarrow{\boldsymbol{a}_1} q_1 \xrightarrow{\boldsymbol{a}_2} \ldots \xrightarrow{\boldsymbol{a}_n} q_n$, where $q_i \in Q$, $\boldsymbol{a}_i \in \boldsymbol{\Sigma}_{q_{i-1}}$, $a_i \in [\![\boldsymbol{a}_i]\!]$, and $\boldsymbol{\delta}(q_{i-1}, \boldsymbol{a}_i) = q_i$ for all $i \in \{1, \cdots, n\}$.

Hence, a symbolic automaton $\boldsymbol{\mathcal{A}}$ can be viewed as an alternative representation of a concrete deterministic automaton whose concrete transition function $\delta : Q \times \Sigma \to Q$ is defined as $\delta(q, a) = \boldsymbol{\delta}(q, \psi_q(a))$. That is, when at q and reading a concrete letter a, the automaton takes the transition $\boldsymbol{\delta}(q, \boldsymbol{a})$, where $\boldsymbol{a}$ is the *unique* element of $\boldsymbol{\Sigma}_q$ satisfying $a \in [\![\boldsymbol{a}]\!]$. The transition function extends to words as in the concrete case. The language recognized by the symbolic automaton, denoted by $L(\boldsymbol{\mathcal{A}})$, consists of all concrete words whose run leads from q_0 to a state in F.

A language $L \subseteq \Sigma$ is *symbolic recognizable* if there exists a symbolic automaton $\boldsymbol{\mathcal{A}}$ such that $L = L(\boldsymbol{\mathcal{A}})$. Let us denote by $SREC(\Sigma)$ the set of all symbolic recognizable languages defined over the large alphabet Σ. One can easily verify that when Σ is a finite alphabet, then $REC(\Sigma) \equiv SREC(\Sigma)$. As for the classical case, two symbolic automata are *equivalent* when they accept the same language.

As mentioned before, the association of a *symbolic language* with a symbolic automaton is more subtle because we allow different partitions of Σ and hence different symbolic alphabets at different states, rendering the transition function *partial* with respect to $\boldsymbol{\Sigma}$. When in a state q and reading a symbol $\boldsymbol{a} \notin \boldsymbol{\Sigma}_q$, the transition to be taken is well defined only when $[\![\boldsymbol{a}]\!] \subseteq [\![\boldsymbol{a}']\!]$ for some $\boldsymbol{a}' \in \boldsymbol{\Sigma}_q$. The model can, nevertheless, be made deterministic and complete over a refinement of the symbolic alphabet. For this, let

$$\boldsymbol{\Sigma}' = \prod_{q \in Q} \boldsymbol{\Sigma}_q, \text{ with the } \Sigma\text{-semantics } [\![(\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n)]\!] = [\![\boldsymbol{a}_1]\!] \cap \ldots \cap [\![\boldsymbol{a}_n]\!]$$

and let $\tilde{\boldsymbol{\Sigma}} = \{\boldsymbol{b} \in \boldsymbol{\Sigma}' : [\boldsymbol{b}] \neq \emptyset\}$. An ordinary automaton $\widetilde{\boldsymbol{\mathcal{A}}} = (\tilde{\boldsymbol{\Sigma}}, Q, \tilde{\delta}, q_0, F)$ can be defined, where, by construction, for every $\boldsymbol{b} \in \tilde{\boldsymbol{\Sigma}}$ and every $q \in Q$, there is $\boldsymbol{a} \in \boldsymbol{\Sigma}_q$ such that $[\![\boldsymbol{b}]\!] \subseteq [\![\boldsymbol{a}]\!]$ and hence one can define the transition function as $\tilde{\boldsymbol{\delta}}(q, \boldsymbol{b}) = \boldsymbol{\delta}(q, \boldsymbol{a})$. This model is more comfortable for language-theoretic studies but we stick hereafter to Definition 4.1 as it is more efficient.

4.3 Operations on Symbolic Automata

After defining and understanding symbolic automata and symbolic recognizable languages, it is important to discuss closure properties as well as decidability of classical problems of this class of languages. In the following, one can find that the set of symbolic recognizable languages is closed under the *Boolean* and *regular* operations. Moreover, decidability problems such as *membership*, *emptiness*, *determinization*, and *equivalence* do not depend on the decidability of the predicate theory that is used and thus they are easily adaptable in the case of symbolic automata [VNG01, HV11, DV14]. Finally, symbolic automata admit a canonical representation which is minimal in the number of states. This is an important

property, especially when inferring symbolic recognizable languages.

Proposition 4.2 (Closure under Boolean Operations)**.** *Languages accepted by deterministic symbolic automata are closed under the Boolean operations.*

Proof. Closure under complementation is immediate by complementing the set of accepting states. For the intersection, we adapt the standard product construction. Then, closure under union follows.

Let L_1, L_2 be two languages recognized, respectively, by the symbolic automata $\boldsymbol{\mathcal{A}}_1 = (\Sigma, \boldsymbol{\Sigma}^1, \psi^1, Q^1, \boldsymbol{\delta}^1, q_0^1, F^1)$ and $\boldsymbol{\mathcal{A}}_2 = (\Sigma, \boldsymbol{\Sigma}^2, \psi^2, Q^2, \boldsymbol{\delta}^2, q_0^2, F^2)$. The intersection of the two languages is then recognized by the symbolic automaton $\boldsymbol{\mathcal{A}} = (\Sigma, \boldsymbol{\Sigma}, \psi, Q, \boldsymbol{\delta}, q_0, F)$, where

- $Q = Q^1 \times Q^2$, $q_0 = (q_0^1, q_0^2)$, $F = F^1 \times F^2$
- For every $(q^1, q^2) \in Q$
  - $\boldsymbol{\Sigma}_{(q^1,q^2)} = \{(\boldsymbol{a}_1, \boldsymbol{a}_2) \in \boldsymbol{\Sigma}^1 \times \boldsymbol{\Sigma}^2 : [\![\boldsymbol{a}_1]\!] \cap [\![\boldsymbol{a}_2]\!] \neq \emptyset\}$
  - $\psi_{(q^1,q^2)}(a) = (\psi^1_{q^1}(a), \psi^2_{q^2}(a))\ \forall a \in \Sigma$
  - $\boldsymbol{\delta}((q^1, q^2), (\boldsymbol{a}_1, \boldsymbol{a}_2)) = (\boldsymbol{\delta}^1(q^1, \boldsymbol{a}_1), \boldsymbol{\delta}^2(q^2, \boldsymbol{a}_2)), \forall (\boldsymbol{a}_1, \boldsymbol{a}_2) \in \boldsymbol{\Sigma}_{(q^1,q^2)}$

It is sufficient to observe that the corresponding implied concrete automata $\mathcal{A}_1, \mathcal{A}_2$ and $\mathcal{A}$ satisfy $\delta((q^1, q^2), a) = (\delta^1(q^1, a), \delta^2(q^2, a))$ and the standard proof that $L(\mathcal{A}) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$ follows. □

One can use the same construction as for the intersection, just by changing the set of final states, to construct a symbolic automaton that recognizes the difference or symmetric difference of two languages. Algorithms for the product and difference constructions for symbolic automata appear in [HV11], as well as experimental evaluation on their performance. Moreover, it is shown that symbolic recognizable languages are closed under the regular operations of concatenations and star [VNG01], following the corresponding constructions for classical finite automata.

A symbolic automaton with ε-transitions is a symbolic automaton that contains transitions of the form $\boldsymbol{\delta}(q, \varepsilon) = q'$, where $\boldsymbol{\varepsilon}$, which is a symbol that represents the empty word, is not a symbol of $\boldsymbol{\Sigma}_q$. An ε-free symbolic automaton is an automaton that does not contain such transitions. Any symbolic automaton can be transformed to an equivalent ε-free symbolic automaton. To eliminate ε-transitions from a symbolic automaton $\boldsymbol{\mathcal{A}}$ the ε-closure method [HMU06] is used, where transitions need to adapt to the symbolic setting [VDHT10].

A symbolic automaton can be made complete simply by adding a sink state q_{sink}. Then, whenever $\bigcup_{\boldsymbol{a} \in \boldsymbol{\Sigma}_q} [\![\boldsymbol{a}]\!] \neq \Sigma$, add a new symbol $\boldsymbol{a}_{sink}$ to $\boldsymbol{\Sigma}_q$, and let $\boldsymbol{\delta}(q, \boldsymbol{a}_{sink}) = q_{sink}$ and $[\![\boldsymbol{a}_{sink}]\!] = \Sigma \setminus \bigcup_{\boldsymbol{a} \in \boldsymbol{\Sigma}_q} [\![\boldsymbol{a}]\!]$.

Determinization of symbolic automata also follows the same idea as for the concrete case. The set of states in the deterministic automaton is a subset of 2^Q. A small but nevertheless important difference is that when we handle symbolic automata the transitions should be defined more carefully not to overlap. To explain this, let us assume non determinism at a state q such that there are two transitions $(q, \boldsymbol{a}, q_1)$ and $(q, \boldsymbol{b}, q_2)$, where $\boldsymbol{a}, \boldsymbol{b} \in \boldsymbol{\Sigma}_q$ and $[\![\boldsymbol{a}]\!] \cap [\![\boldsymbol{b}]\!] \neq \emptyset$. To make the state deterministic we replace the involved states by new states, and replace the transitions by the new ones $(\{q\}, \boldsymbol{a}_1, \{q_1\})$, $(\{q\}, \boldsymbol{a}_{12}, \{q_1, q_2\})$, and $(\{q\}, \boldsymbol{a}_2, \{q_2\})$, where $[\![\boldsymbol{a}_1]\!] = [\![\boldsymbol{a}]\!] \setminus [\![\boldsymbol{b}]\!]$, $[\![\boldsymbol{a}_{12}]\!] = [\![\boldsymbol{a}]\!] \cap [\![\boldsymbol{b}]\!]$, and $[\![\boldsymbol{a}_2]\!] = [\![\boldsymbol{b}]\!] \setminus [\![\boldsymbol{a}]\!]$, as long as the semantics remain non empty.

Finally, symbolic automata admit a canonical representation. For any symbolic recognizable language, there exists a unique minimal automaton recognizing it, up to isomorphism on the states. As in the classical case, we can define a Nerode equivalence relation and then construct the minimal automaton.

The left quotient can be extended to a symbolic prefix $\boldsymbol{u} \in \boldsymbol{\Sigma}^*$ as follows:

$$\boldsymbol{u}^{-1}L = \{v \in \Sigma^* : uv \in L, u \in [\![\boldsymbol{u}]\!]\}.$$

A Nerode equivalence relation, induced by the language $L \subseteq \Sigma^*$, can be lifted to symbolic words as follows

$$\boldsymbol{u} \equiv_L \boldsymbol{v} \text{ if and only if } \boldsymbol{u}^{-1}L = \boldsymbol{v}^{-1}L.$$

Based on the Nerode Theorem (Theorem 2.1) we build a minimal symbolic automaton $\boldsymbol{\mathcal{A}}$ using the syntactic congruence. We define this automaton as $\boldsymbol{\mathcal{A}} = (\Sigma, \boldsymbol{\Sigma}, Q, q_0, \boldsymbol{\delta}, F)$, where Q is the set of left quotients of L, i.e., $Q = \Sigma^*/\equiv_L$; the initial state is the class that contains the empty word, i.e., $q_0 = [\varepsilon]$; and final states are all those classes that contain accepted words, i.e., $F = \{[u] : u \cdot \varepsilon \in L\}$. Defining the symbolic letters is a bit more tricky and only symbols with non-empty semantics are allowed. Let p and q be two states from Q that correspond to the classes $[u]$ and $[v]$ respectively, we define $\boldsymbol{\Sigma}_p$ as the set of letters $\boldsymbol{a}_{pq}$ whose semantics is defined as $[\![\boldsymbol{a}_{pq}]\!] = \{a \in \Sigma : [u \cdot a] \equiv_L [v]\}$ and it is not empty. The symbolic transition function is then given by $\boldsymbol{\delta}(p, \boldsymbol{a}_{pq}) = q$. By construction, this automaton has no redundant symbols, moreover, it is complete and deterministic. Therefore, $\boldsymbol{\mathcal{A}}$ is minimal [DV14].

Moreover, $\boldsymbol{\mathcal{A}}$ is unique up to renaming of states and up to the equivalence of the Σ-semantics of the symbols. That is, if p, q are two states in $\boldsymbol{\mathcal{A}}$, and p', q' are two states in $\boldsymbol{\mathcal{B}}$, which is another minimal automaton that recognizes L, such that $p \sim p'$ and $q \sim q'$, then $[\![\boldsymbol{a}_{pq}]\!] = [\![\boldsymbol{b}_{p'q'}]\!]$ where $\boldsymbol{a}_{pq}$ and $\boldsymbol{b}_{p'q'}$ are symbolic letters in $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$, respectively.

Algorithms for minimization of symbolic automata can be found in [DV14]. In this paper, the authors present an adaptation of the classical minimization algorithms, such as Moore's and Hopcroft's algorithms, to the symbolic setting as well as a symbolic minimization algorithm.

4.4 Alphabets and Partitions

Symbolic automata, as defined in the two previous sections, can be defined over any type of alphabet and the predicates can be of arbitrary form. However, we restrict ourselves to predicates that form a partition over the alphabet at each state, and thus to deterministic symbolic automata. This section presents some examples of symbolic automata over specific alphabet domains and partitions. Details on sets and partitions can be found in Section 2.2

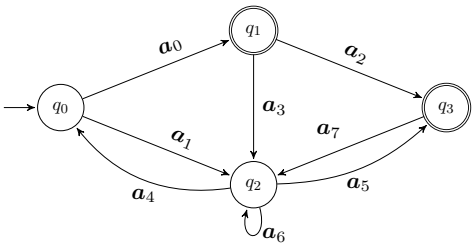
4.4.1 Interval Automata

In this section, we assume an ordered alphabet that is a finite or infinite subset of a numerical set such as $\mathbb{N}$, $\mathbb{Q}$, or $\mathbb{R}$. We let the Σ-semantics of each symbol form closed-open intervals. To preserve convexity[1], unions of intervals are not allowed in the semantics. This practice is preferable due to simplicity, even though it may cause the symbolic automaton to not be minimal with respect to the number of transitions, as defined in Section 4.3. Assuming a small number of intervals in each partition, the symbolic automaton can still be well defined and have a finite symbolic alphabet. In the following, we provide some concrete examples.

Example 4.3. Let $\mathcal{A}$ be an interval symbolic automaton as the one shown in Figure 4.1. The symbolic transition function $\boldsymbol{\delta}$ is finite and can be given as a table. The symbolic alphabet $\boldsymbol{\Sigma} = \{\boldsymbol{a}_0, \ldots, \boldsymbol{a}_7\}$ can also be viewed as the union of the state-depended symbolic alphabets $\boldsymbol{\Sigma}_{q_0} = \{\boldsymbol{a}_0, \boldsymbol{a}_1\}$, $\boldsymbol{\Sigma}_{q_1} = \{\boldsymbol{a}_2, \boldsymbol{a}_3\}$, $\boldsymbol{\Sigma}_{q_2} = \{\boldsymbol{a}_4, \boldsymbol{a}_5, \boldsymbol{a}_6\}$, and $\boldsymbol{\Sigma}_{q_3} = \{\boldsymbol{a}_7\}$. The concrete input alphabet of $\mathcal{A}$ is $\Sigma = [0, 100) \subseteq \mathbb{R}$. The Σ-semantics for the symbols are defined as $[\![\boldsymbol{a}_0]\!] = [0, 50)$, $[\![\boldsymbol{a}_1]\!] = [50, 100)$, $[\![\boldsymbol{a}_2]\!] = [0, 30)$, etc., see Figure 4.2.

A similar symbolic automaton can be defined by letting the concrete input alphabet be a subset of the natural numbers $\mathbb{N}$, that is $\Sigma = \{0, 1, \ldots, 99\}$. This automaton has the same structure as $\mathcal{A}$ and differs only in its Σ-semantics, which should be intersected with $\mathbb{N}$ for all $\boldsymbol{a}_i \in \boldsymbol{\Sigma}$, e.g., $[\![\boldsymbol{a}_0]\!] = \{0, \ldots, 49\}$, $[\![\boldsymbol{a}_1]\!] = \{50, \ldots, 99\}$, $[\![\boldsymbol{a}_2]\!] = \{0, \ldots, 29\}$, etc.

[1] A subset Σ of $\mathbb{R}^n$ is said to be *convex* if it contains all the line segments connecting any pair of its points. Every interval is, by definition, a convex set.



(a)

δ	q_0	q_1	q_2	q_3
q_0	–	a_0	a_1	–
q_1	–	–	a_3	a_2
q_2	a_4	–	a_6	a_5
q_3	–	–	a_7	–

(b)

Figure 4.1: A symbolic automaton $\mathcal{A}$ with its symbolic transition function δ.

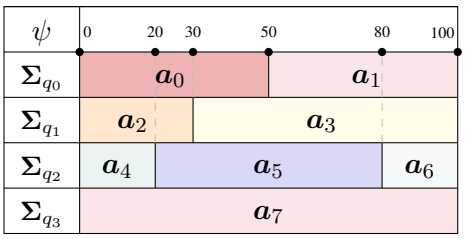


Figure 4.2: The concrete semantics of the symbols of automaton $\mathcal{A}$ of Figure 4.1. The input alphabet is $\Sigma = [0, 100) \subseteq \mathbb{R}$.

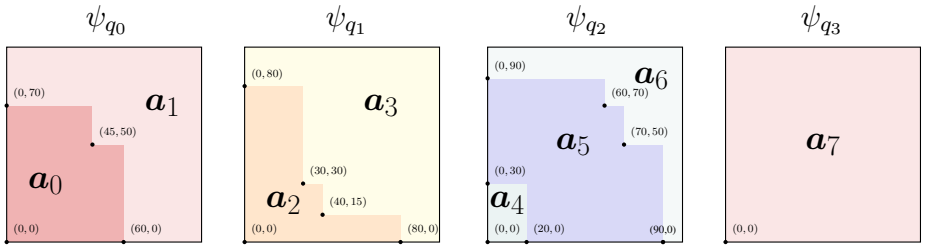


Figure 4.3: The monotone partitions over the concrete alphabet $\Sigma = [0, 100)^2$ as defined by ψ_q at each state q of the automaton $\mathcal{A}$ (Figure 4.1).

4.4.2 Automata over Partially-ordered Alphabets

In this section we assume the input alphabet to be a partially-ordered set of the form $\Sigma = X^d$, where X is a totally-ordered set such as an interval $[0, k) \subseteq \mathbb{R}$. We can define a symbolic automaton $\boldsymbol{\mathcal{A}}$ where at each state q the function ψ_q forms a monotone partition of the concrete alphabet defined as a finite union of cones, see Section 2.2. Such a partition is of the form $\mathcal{P} = \{P_1, \ldots, P_{m-1}\}$, where $P_i = B^+(F_{i-1}) - B^+(F_i)$ and $\mathcal{F} = \{F_0, \ldots, F_{m-1}\}$ is a family of sets of minimal incomparable points in Σ. Each symbol $\boldsymbol{a}_i \in \boldsymbol{\Sigma}_q$ associates to the partition block P_i, such that $[\![\boldsymbol{a}_i]\!] = P_i$. We assume that partition blocks are simply-connected and the Σ-semantics does not form unions of non-connected subsets of Σ.

Example 4.4. Let the alphabet be $\Sigma = [0, 100)^2$. A symbolic automaton $\boldsymbol{\mathcal{A}}$ with a monotone partition at each state is shown in Figure 4.1. The semantics function ψ, which is shown in Figure 4.3, forms at each state a monotone partition of Σ. The partitions are defined from the following families of points:

$$P_{q_0} = \big\{\{(0,0)\}, \{(0,70), (45,50), (60,0)\}\big\},$$
$$P_{q_1} = \big\{\{(0,0)\}, \{(0,80), (30,30), (40,15), (80,0)\}\big\},$$
$$P_{q_2} = \big\{\{(0,0)\}, \{(0,30), (20,0)\}, \{(0,90), (60,70), (70,50), (90,0)\}\big\}, \text{and}$$
$$P_{q_3} = \big\{\{(0,0)\}\big\}.$$

4.4.3 Boolean Vectors

Let us define a symbolic automaton $\boldsymbol{\mathcal{A}}$ over the concrete alphabet $\Sigma = \mathbb{B}^n$, that is, the Boolean hyper-cube of dimension $n \in \mathbb{N}$ where letters are vectors accessed by the Boolean variables $X = \{x_1, \ldots, x_n\}$. We let the Σ-semantics of the symbolic letters form sub-cubes of Σ, represented by a family of functions $\{\psi_q\}_{q \in Q}$ as usual. Hence, for each state q of a symbolic automaton, the function ψ_q is a *pseudo-boolean function* $\psi_q : \mathbb{B}^n \to \boldsymbol{\Sigma}_q$. There are multiple ways to represent such a function. We choose the representations of a *binary decision tree* (BDT), a *pseudo-boolean normal form* (PBNF), or Karnaugh maps when the number of variables n is small. We refer the reader to Section 2.2 for a brief description of partitions over a Boolean Cube and its different representations.

Example 4.5. Let the symbolic automaton $\boldsymbol{\mathcal{A}}$ have the same structure as the one of Figure 4.1. Transitions are taken based on the same transition function but the semantics are determined by a family $\psi = \{\psi_q\}_{q \in Q}$ of pseudo-boolean functions.

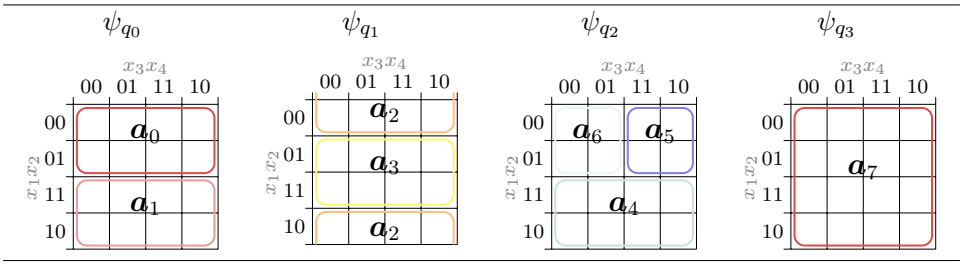


Figure 4.4: The partitions over the concrete alphabet $\Sigma = \mathbb{B}^4$ as defined by ψ_q at each state q of the automaton $\boldsymbol{\mathcal{A}}$ (Figure 4.5).

The Σ-semantics, which are shown in Figure 4.4 as Karnaugh maps, are defined as

$$\begin{aligned}
\psi_{q_0} &= \boldsymbol{a}_0 \cdot \bar{x_1} + \boldsymbol{a}_1 \cdot x_1, \\
\psi_{q_1} &= \boldsymbol{a}_2 \cdot \bar{x_2} + \boldsymbol{a}_3 \cdot x_2, \\
\psi_{q_2} &= \boldsymbol{a}_4 \cdot x_1 + \boldsymbol{a}_5 \cdot \bar{x_1}x_3 + \boldsymbol{a}_6 \cdot \bar{x_1}\bar{x_3}, \text{and} \\
\psi_{q_3} &= \boldsymbol{a}_7.
\end{aligned}$$

The symbolic automaton $\boldsymbol{\mathcal{A}}$ can be seen in Figure 4.5 where the symbols are replaced by Boolean guards.

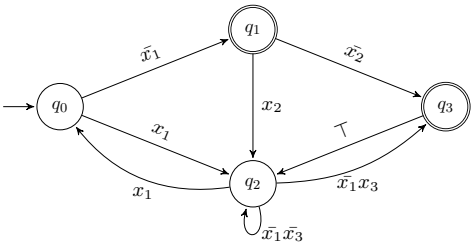


Figure 4.5: The symbolic automaton $\boldsymbol{\mathcal{A}}$ of Figure 4.1 where a symbolic transition is taken when a guard, which is a Boolean function, is satisfied.

5 CHAPTER

Learning Symbolic Automata

Learning algorithms, as presented in Chapter 3, infer an automaton from a finite set of words (the sample) for which membership is known. Over small alphabets, the sample should include the set of the shortest words that lead to each state and, in addition, the set of all their Σ-continuations, where Σ is the input alphabet. Over large alphabets, these algorithms do not scale well. As an alternative, we develop a symbolic learning algorithm over symbolic words, which are only partially backed up by the sample. In a sense, our algorithm is a combination of automaton learning and learning of non-temporal predicates.

In this chapter, we introduce the *symbolic learning algorithm*, an algorithm for learning languages over large alphabets that can be represented by symbolic automata models (see Chapter 4). Our algorithm can be seen as a symbolic adaptation of the L^* algorithm. We assume an oracle, the *teacher*, that knows the target language and can answer *membership* (MQ) and *equivalence queries* (EQ). The *learner* first asks MQ's and collects all the information in a table structure, the *symbolic observation table*, and then, when the information in the symbolic observation table is sufficient, the learner constructs a hypothesis. The hypothesis is tested by posing an EQ; whenever the teacher returns a counter-example, the learner revises the hypothesis, it terminates otherwise.

The entries of the symbolic observation table constitute statements about the inclusion or exclusion of a large set of concrete words in the language. To prevent asking MQ's concerning all concrete words, only a small subset of words is chosen, the *evidences*; to avoid exponential growth of the table, only one of them, the *representative*, will be used for subsequent queries.

Handling multiple evidences to determine partitions is a crucial feature in learning automata over large alphabets. Evidences of the same partition should behave the same; if they do not, this causes evidence incompatibility. Finding and resolving evidence incompatibility is a major novel feature that is used in symbolic

learning algorithms, as presented in this chapter.

However, parts of the algorithm can be adapted to the particularities of a specific learning problem, for the algorithm to become more efficient or result in a better hypothesis. For instance, the inferred model cannot be precise when the equivalence query is approximated. Furthermore, the input alphabet of the target language, or knowledge of the nature of the partitions, affects the way we learn transitions. Specialized methods to select evidences may result in more precise conjectures.

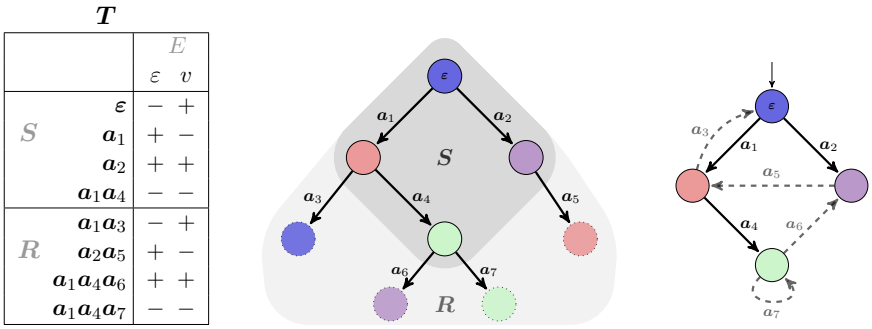
We are not the first to study the application of automaton learning to large alphabets. In order to make a comparison between related work and ours meaningful, we postpone it to Section 5.2, after the relevant notions have been introduced. In Section 5.1, we give formal definitions of symbolic observation tables and their properties, define evidences and representatives, and explain the notion of evidence incompatibility. The main algorithm and procedures follow and are presented in a general form in Section 5.3. The instantiation of the algorithm to specific alphabets and equivalence checking methods are discussed separately in Chapters 6 and 7, respectively.

5.1 Definitions

To identify states and transitions and finally conjecture a hypothesis, the learner organizes all the examples and information that is available in a data-structure that is called the *symbolic observation table*. The rows of such a table correspond to symbolic words, which represent access sequences to the states; the columns consist of concrete words that are used to distinguish the states; and the entries of the table are instances of the symbolic characteristic function of the target language, which are incrementally filled in after posing MQ's to the teacher. Individual symbols represent transitions and need not be the same in each state. Each symbol has its own semantics, evidences and representatives.

Learning an automaton is like growing a spanning tree, where nodes stand for states and edges are transitions. With the initial state placed at the root of the tree, it is sufficient to grow the tree until all leaf nodes correspond to states that have been already reached via another path in the tree. In the following, we present such trees, which we call *balanced trees*, and relate them to the symbolic observation tables.

The structure of a balanced tree appears in Figure 5.1b together with its corresponding automaton in Figure 5.1c. The underlying intuition is that elements of $\boldsymbol{S}$, also known as access sequences, correspond to a spanning tree of the transition graph of the automaton to be learned, while elements of the boundary $\boldsymbol{R}$ correspond to back- and cross-edges relative to this spanning tree.



(a) (b) (c)

Figure 5.1: (a) A symbolic observation table, (b) its balanced symbolic Σ-tree, and (c) the conjectured automaton.

Let Σ be a large input alphabet; let $\boldsymbol{S} \uplus \boldsymbol{R}$ be the set of access sequences, a prefix-closed subset of $\boldsymbol{\Sigma}^*$, where $\boldsymbol{\Sigma}$ is a symbolic alphabet such that $\boldsymbol{\Sigma} = \biguplus_{\boldsymbol{s} \in \boldsymbol{S}} \boldsymbol{\Sigma}_{\boldsymbol{s}}$; and, let $\psi = \{\psi_{\boldsymbol{s}}\}_{\boldsymbol{s} \in \boldsymbol{S}}$ be a family of total surjective functions of the form $\psi_{\boldsymbol{s}} : \Sigma \to \boldsymbol{\Sigma}_{\boldsymbol{s}}$, that defines the semantics of the symbols. A *balanced symbolic Σ-tree* is a tuple $(\boldsymbol{\Sigma}, \boldsymbol{S}, \boldsymbol{R}, \psi)$, where for every $\boldsymbol{s} \in \boldsymbol{S}$ and $\boldsymbol{a} \in \boldsymbol{\Sigma}_{\boldsymbol{s}}$, $\boldsymbol{s} \cdot \boldsymbol{a} \in \boldsymbol{S} \cup \boldsymbol{R}$, and for any $\boldsymbol{r} \in \boldsymbol{R}$ and $\boldsymbol{a} \in \boldsymbol{\Sigma}$, $\boldsymbol{r} \cdot \boldsymbol{a} \notin \boldsymbol{S} \cup \boldsymbol{R}$. Elements of $\boldsymbol{R}$ are called *boundary elements* of the tree.

Definition 5.1 (Symbolic Observation Table)**.** A *symbolic observation table* is a tuple $\boldsymbol{T} = (\Sigma, \boldsymbol{\Sigma}, \boldsymbol{S}, \boldsymbol{R}, \psi, E, \boldsymbol{f}, \mu, \hat{\mu})$ such that

- Σ is an alphabet,
- $(\boldsymbol{\Sigma}, \boldsymbol{S}, \boldsymbol{R}, \psi)$ is a balanced symbolic Σ-tree,
- $E \subseteq \Sigma^*$ is a set of *distinguishing words*,
- $\boldsymbol{f} : (\boldsymbol{S} \cup \boldsymbol{R}) \times E \to \{-, +\}$ is the *symbolic classification function*,
- $\mu : \boldsymbol{\Sigma} \to 2^{\Sigma} - \{\emptyset\}$ is the *evidence function*, where $\mu(\boldsymbol{a}) \subseteq [\![\boldsymbol{a}]\!]$ for all $\boldsymbol{a} \in \boldsymbol{\Sigma}$,
- $\hat{\mu} : \boldsymbol{\Sigma} \to \Sigma$ is the *representative function*, where $\hat{\mu}(\boldsymbol{a}) \in \mu(\boldsymbol{a})$ for all $\boldsymbol{a} \in \boldsymbol{\Sigma}$.

The evidence and representative functions are extended to symbolic words in $\boldsymbol{S} \cup \boldsymbol{R}$ as follows:

$$\begin{aligned} \mu(\boldsymbol{\varepsilon}) &= \{\varepsilon\} & \hat{\mu}(\boldsymbol{\varepsilon}) &= \varepsilon \\ \mu(\boldsymbol{s} \cdot \boldsymbol{a}) &= \hat{\mu}(\boldsymbol{s}) \cdot \mu(\boldsymbol{a}) & \hat{\mu}(\boldsymbol{s} \cdot \boldsymbol{a}) &= \hat{\mu}(\boldsymbol{s}) \cdot \hat{\mu}(\boldsymbol{a}). \end{aligned} \tag{5.1}$$

The symbolic characteristic function values are based on the representative of the symbolic prefix rather than the set of all evidences, i.e., to fill the $(\boldsymbol{s}, e)$ entry in the table we let $\boldsymbol{f}(\boldsymbol{s}, e) = f(\hat{\mu}(\boldsymbol{s}) \cdot e)$ where f is the characteristic function of the target language. By this, an undesirable growth in the size of the queries and size

of the table is avoided. With every $\boldsymbol{s} \in \boldsymbol{S} \cup \boldsymbol{R}$ we associate a residual classification function $\boldsymbol{f_s} : E \to \{-, +\}$ defined as $\boldsymbol{f_s}(e) = \boldsymbol{f}(\boldsymbol{s}, e)$, $e \in E$.

The *symbolic sample* associated with $\boldsymbol{T}$ is the set $\boldsymbol{M_T} = (\boldsymbol{S} \cup \boldsymbol{R}) \cdot E$ while the *concrete sample*, the set of all words whose classification is known, is $M_{\boldsymbol{T}} = \mu(\boldsymbol{S} \cup \boldsymbol{R}) \cdot E$, and coincides with the domain of f. Note that for each word $\boldsymbol{w} \in \boldsymbol{M_T}$ there is at least one concrete $w \in \mu(\boldsymbol{w})$ whose membership in L is known.

Let $\mu_{\boldsymbol{s}} = \bigcup_{\boldsymbol{a} \in \boldsymbol{\Sigma_s}} \mu(\boldsymbol{a})$ be the set of all evidences for a state $\boldsymbol{s} \in \boldsymbol{S}$. Evidences of the same symbol should behave the same and when this is not the case, that is, when two concrete letters in the evidence of a symbol lead to different residual functions, we call this a manifestation of *evidence incompatibility*. Evidence incompatibility can be characterized and measured as follows.

Definition 5.2 (Incompatibility Instance)**.** A state $\boldsymbol{s} \in \boldsymbol{S}$ has an *incompatibility instance* at evidence $a \in \mu_{\boldsymbol{s}}$ when $f_{\hat{\mu}(\boldsymbol{s}) \cdot a} \neq f_{\hat{\mu}(\boldsymbol{s} \cdot \boldsymbol{a})}$, and $\psi_{\boldsymbol{s}}(a) = \boldsymbol{a}$; this fact is denoted by $\textsc{inc}(\boldsymbol{s}, a)$, where

$$\textsc{inc}(\boldsymbol{s}, a) = \begin{cases} 1, & \text{if } f_{\hat{\mu}(\boldsymbol{s}) \cdot a} \neq f_{\hat{\mu}(\boldsymbol{s}) \cdot \hat{\mu}(\psi_{\boldsymbol{s}}(a))} \\ 0, & \text{otherwise} \end{cases}$$

Definition 5.3 (Incompatibility Degree)**.** The *evidence incompatibility degree* associated to a state $\boldsymbol{s} \in \boldsymbol{S}$ is the total number of incompatibility instances that the state $\boldsymbol{s}$ has. We denote this by $\mathrm{M}(\boldsymbol{s})$, where $\mathrm{M}(\boldsymbol{s}) = |\{a \in \mu_{\boldsymbol{s}} : \textsc{inc}(\boldsymbol{s}, a)\}|$.

Naturally, the evidence incompatibility of a table $\boldsymbol{T}$ is the sum of the incompatibility degree over all its states $\boldsymbol{s} \in \boldsymbol{S}$, i.e., $\mathrm{M}(\boldsymbol{T}) = \sum_{\boldsymbol{s} \in \boldsymbol{S}} \mathrm{M}(\boldsymbol{s})$. Note that the incompatibility degree is bounded by the number of states times the maximum number of evidences per state.

Definition 5.4 (Table Properties)**.** A table $\boldsymbol{T} = (\Sigma, \boldsymbol{\Sigma}, \boldsymbol{S}, \boldsymbol{R}, \psi, E, \boldsymbol{f}, \mu, \hat{\mu})$ is

- *Closed* if $\forall \boldsymbol{r} \in \boldsymbol{R}$, $\exists \boldsymbol{s} \in \boldsymbol{S}$, $\boldsymbol{f_r} = \boldsymbol{f_s}$,
- *Reduced* if $\forall \boldsymbol{s}, \boldsymbol{s'} \in \boldsymbol{S}$, $\boldsymbol{f_s} \neq \boldsymbol{f_{s'}}$, and
- *Evidence compatible* if $\mathrm{M}(\boldsymbol{s}) = 0, \forall \boldsymbol{s} \in \boldsymbol{S}$.

The properties of the table, as defined above, are crucial to hold when one wants to conjecture a target language and build a symbolic automaton compatible with the concrete sample. A reduced table does not include any redundant prefixes and thus restricts the set of states to be exactly $\boldsymbol{S}$. A closed table leads to a well defined transition function. Finally, evidences should be compatible and behave as their representatives for the automaton to be compatible with $M_{\boldsymbol{T}}$.

The following result is the natural generalization of the derivation of an automaton from an observation table to the symbolic setting, see Theorem 3.3. Figure 5.1 illustrates the connection of the observation table to an automaton.

Theorem 5.5 (Automaton from Table)**.** *From a closed, reduced and evidence compatible table one can construct a deterministic symbolic automaton compatible with the concrete sample.*

Proof. Let $\boldsymbol{T} = (\Sigma, \boldsymbol{\Sigma}, \boldsymbol{S}, \boldsymbol{R}, \psi, E, \boldsymbol{f}, \mu, \hat{\mu})$ be a symbolic observation table that is closed, reduced, and evidence compatible. We define a function $g : \boldsymbol{R} \to \boldsymbol{S}$, such that $g(\boldsymbol{r}) = \boldsymbol{s}$ if and only if $\boldsymbol{f_r} = \boldsymbol{f_s}$. The automaton derived from the table is then $\boldsymbol{\mathcal{A}_T} = (\Sigma, \boldsymbol{\Sigma}, \psi, Q, \boldsymbol{\delta}, q_0, F)$ where:

- $Q = \boldsymbol{S}$, $q_0 = \boldsymbol{\varepsilon}$
- $F = \{\boldsymbol{s} \in \boldsymbol{S} : \boldsymbol{f_s}(\varepsilon) = +\}$
- $\boldsymbol{\delta} : Q \times \boldsymbol{\Sigma} \to Q$ is defined as $\boldsymbol{\delta}(\boldsymbol{s}, \boldsymbol{a}) = \begin{cases} \boldsymbol{s} \cdot \boldsymbol{a}, & \text{when } \boldsymbol{s} \cdot \boldsymbol{a} \in \boldsymbol{S} \\ g(\boldsymbol{s} \cdot \boldsymbol{a}), & \text{when } \boldsymbol{s} \cdot \boldsymbol{a} \in \boldsymbol{R} \end{cases}$

By construction and like in [Ang87], $\boldsymbol{\mathcal{A}_T}$ classifies correctly via $\boldsymbol{f}$ the symbolic sample and, due to evidence compatibility, this classification agrees with the characteristic function f on the concrete sample. □

5.2 Comparison to Related Work

To start with, our work on symbolic automata should not be confused with work dealing with register automata, a popular extension of automata to infinite alphabets [KF94, BLP10, HSJC12]. These are automata augmented with additional variables that can store in registers some input letters encountered while reading a word. Newly-read letters can be compared with the registers but typically not with constants in the domain. With register automata one can express, for example, the requirement that the password at login is the same as the password at sign-up. In the most recent work on learning register automata [CHJS16], a strong tree oracle is used. Given a concrete prefix and a symbolic prefix, the teacher returns a special type of a register automaton that has a tree structure. This fills in the entries of the observation table and provides the information about the registers and guards in the automaton. This algorithm is efficient only in the presence of shortest counter-examples and, in addition, when applied to a theory of inequalities and extended to use constants, these constants should be known in advance.

Our approach, which uses symbolic automata, is different. First, no auxiliary variables are added to the automaton. The values of the input symbols influence transitions via predicates, possibly of a restricted complexity. These predicates involve domain constants and they partition the alphabet into finitely many classes. For example, over the integers, a state may have transitions labeled by conditions

of the form $c_1 \leq x \leq c_2$, which give real (but of limited resolution) access to the input domain. On the other hand, we insist on finite (and small) memory so that the exact value of x cannot be registered and has no future influence beyond the transition it has triggered. Many control systems, artificial (sequential machines working on quantized numerical inputs) as well as natural (central nervous system, the cell), are believed to operate in this manner.

Ideas similar to ours have been suggested and explored in a series of papers [BJR06, HSM11, IHS13] that also adapt automaton learning and Angluin's algorithm to large alphabets and are the closest conceptually to our work. These papers have some design decisions which are similar, for example, the usage of distinct symbolic alphabets at every state [IHS13], the notion of evidence or the adaptation of the breakpoint method [HSM11].

In contrast to our learning algorithm, the scheme presented in [BJR06] is always based on alphabet refinement and has the potential of generating new symbols indefinitely. On the other hand, the algorithms in [HSM11, IHS13] result in a partially defined hypothesis, where the transition function is not defined outside the observed evidence.

The main distinguishing feature in our framework is that it is more rigorous and comprehensive in the way it treats evidence incompatibility and the modification of partition boundaries. We do not consider each modification as a partition refinement, but rather try first to modify the boundaries and only add a new symbol when necessary. As a result, we have the following properties, whenever we conclude the treatment of evidence incompatibility. First, the number of symbolic letters is never larger than the minimal number needed; and second, the mapping of concrete letters to symbols is always sample-compatible and it is well-defined for the whole concrete alphabet.

Recently, new results presented in [DD17] in the context of learning symbolic automata give a more general justification for a learning scheme like ours by proving that learnability is closed under product and disjoint union.

Moreover, some similarities can also be expected between our symbolic algorithm and the S^* algorithm in [BB13]. However, this work seems to be incomparable to ours as they use a richer model of transducers as well as more general predicates on inputs and outputs. Their termination result is weaker and is closely related to the termination of the counter-example guided abstraction refinement procedure.

To conclude, we believe that our algorithm for learning languages over large alphabets is unique in employing all the following features:

1. It is based on a clean and general definition of the relation between the concrete and symbolic alphabets;

2. It treats the modification of alphabet partitions in a rigorous way which guarantees that no superfluous symbols are introduced;
3. It accommodates for a teacher free setting, where counter-examples need not be minimal, and PAC learnability can be achieved;
4. It employs an adaptation of the breakpoint method to analyze in an efficient way the information provided by counter-examples;
5. It is modular, separating the general aspects from those that are alphabet specific, thus providing for a relatively easy adaptation to different alphabets.

5.3 The Symbolic Learning Algorithm

In this section, we present the symbolic learning algorithm in a general and abstract way that does not assume any structure on the alphabet. Moreover, the type of equivalence checking is not specified. As one might notice, some of the procedures and methods cannot be fully specified in this general setting. The input alphabet, the nature of the partitions, and the type of equivalence oracle are all important elements that can affect the way transitions are learned, the efficiency of the algorithm, and sometimes, even its termination.

The main sampling-based symbolic learning algorithm, summarized in Algorithm 8, alternates between two phases. In the first phase it attempts to make the table closed (line 5) and evidence compatible (line 6) so as to construct a symbolic automaton (line 8). In the second phase, after formulating an equivalence query (line 9), it processes the provided counter-example (line 12), which renders the table not closed or evidence incompatible. These phases alternate until no counter-example is found. Note that the table, by construction, is always kept reduced. The procedures that appear in Algorithm 8 are explained in detail hereafter.

Whenever MQ is invoked for word w, then w is added to the concrete sample and the characteristic function f is updated such that $f(w) = \text{MQ}(w)$.

Table Initialization (Procedure 9). The algorithm builds an initial observation table $\boldsymbol{T}$, with $\boldsymbol{\Sigma}_\varepsilon = \{\boldsymbol{a}\}$, $\boldsymbol{S} = \{\boldsymbol{\varepsilon}\}$, $\boldsymbol{R} = \{\boldsymbol{a}\}$, $E = \{\varepsilon\}$. The newly introduced symbol $\boldsymbol{a}$ is initialized with concrete semantics, evidences and a representative, via the procedure INITSYMBOL which is invoked each time a new state is introduced. Then membership queries are posed to update $\boldsymbol{f}$ and fill the table.

Symbol Initialization (Procedure 10). For a new symbolic letter $\boldsymbol{a}$, we let $[\![\boldsymbol{a}]\!] = \Sigma$ and as an evidence $\mu(\boldsymbol{a})$ we take a set of k concrete letters, denoted by $sample(\Sigma, k)$. One element of the evidence, denoted by $select(\mu(\boldsymbol{a}))$, is chosen as a representative and will be used to fill table entries for all rows where $\boldsymbol{a}$ appears.

Algorithm 8 Symbolic Learning Algorithm

$learned = false$
INITTABLE($\boldsymbol{T}$)
repeat
  while $\boldsymbol{T}$ is not closed **or** not evidence compatible **do**
    CLOSE
    EVCOMP
  end while
  Construct $\mathcal{A}_{\boldsymbol{T}}$
  if EQ($\mathcal{A}_{\boldsymbol{T}}$) **then** ▷ check hypothesis $\mathcal{A}_{\boldsymbol{T}}$
    $learned = true$
  else ▷ a counter-example w is provided
    COUNTEREX($\mathcal{A}_{\boldsymbol{T}}, w$) ▷ process counter-example
  end if
until $learned$
return $\mathcal{A}_{\boldsymbol{T}}$

Procedure 9 Table Initialization

procedure INITTABLE($\boldsymbol{T}$)
  $\boldsymbol{\Sigma_\varepsilon} = \{\boldsymbol{a}\}$; $\boldsymbol{S} = \{\boldsymbol{\varepsilon}\}$; $\boldsymbol{R} = \{\boldsymbol{a}\}$; $E = \{\varepsilon\}$ ▷ $\boldsymbol{a}$ is a new symbol
  INITSYMBOL($\boldsymbol{a}$)
  Ask MQ(u) for all $u \in \mu(\boldsymbol{a}) \cup \{\varepsilon\}$
  $\boldsymbol{f}(\boldsymbol{\varepsilon}) = f(\varepsilon)$; $\boldsymbol{f}(\boldsymbol{a}) = f(\hat{\mu}(\boldsymbol{a}))$
  $\boldsymbol{T} = (\Sigma, \boldsymbol{\Sigma}, \boldsymbol{S}, \boldsymbol{R}, \psi, E, \boldsymbol{f}, \mu, \hat{\mu})$
end procedure

The $sample(\Sigma, k)$ or $select$ methods can use a uniform distribution, but one can think of evidence or representative selection as a more adaptive process that may depend on the outcome of membership queries or other assumptions we have made, e.g., the minimality on the counter-examples, a case explained in more detail in Section 6.1, allows us to select as evidence and representative always the minimal element of a partition.

Table Closing (Procedure 11). A table is not closed when there exists some $\boldsymbol{r} \in \boldsymbol{R}$ without any equivalent element $\boldsymbol{s} \in \boldsymbol{S}$ such that $\boldsymbol{f_r} = \boldsymbol{f_s}$. To render the table closed, $\boldsymbol{r}$ should be considered as a new state. To this end, $\boldsymbol{r}$ is moved from $\boldsymbol{R}$ to $\boldsymbol{S}$ with alphabet $\boldsymbol{\Sigma_r} = \{\boldsymbol{a}\}$, where $\boldsymbol{a}$ is a new symbol which is initialized. To balance the table the new word $\boldsymbol{r} \cdot \boldsymbol{a}$ is added to $\boldsymbol{R}$, its evidence $\mu(\boldsymbol{r} \cdot \boldsymbol{a})$ and representative $\hat{\mu}(\boldsymbol{r} \cdot \boldsymbol{a})$ are computed following (5.1) and membership queries are

Procedure 10 Symbol Initialization

procedure INITSYMBOL($\boldsymbol{a}$)
  $[\![\boldsymbol{a}]\!] = \Sigma$
  $\mu(\boldsymbol{a}) = sample(\Sigma, k)$
  $\hat{\mu}(\boldsymbol{a}) = select(\mu(\boldsymbol{a}))$
end procedure

Procedure 11 Table Closing

procedure CLOSE
  Given $\boldsymbol{r} \in \boldsymbol{R}$ such that $\forall \boldsymbol{s} \in \boldsymbol{S}$, $\boldsymbol{f_r} \neq \boldsymbol{f_s}$
  $\boldsymbol{S} = \boldsymbol{S} \cup \{\boldsymbol{r}\}$ ▷ declare $\boldsymbol{r}$ a new state
  $\boldsymbol{\Sigma_r} = \{\boldsymbol{a}\}$ ▷ introduce a new symbol $\boldsymbol{a}$
  INITSYMBOL($\boldsymbol{a}$)
  $\boldsymbol{R} = (\boldsymbol{R} - \{\boldsymbol{r}\}) \cup \{\boldsymbol{r} \cdot \boldsymbol{a}\}$ ▷ add new boundary element
  Ask MQ(u) for all $u \in \mu(\boldsymbol{r} \cdot \boldsymbol{a}) \cdot E$
  $\boldsymbol{f_{r \cdot a}} = f_{\hat{\mu}(\boldsymbol{r} \cdot \boldsymbol{a})}$
end procedure

posed to update f and fill the table.

Fixing Evidence Incompatibility (Procedure 12). A table is not evidence compatible when the evidence incompatibility degree of a state $\boldsymbol{s} \in \boldsymbol{S}$ is greater than zero. This happens when there exist a letter $a \in \mu_{\boldsymbol{s}}$ and a suffix $e \in E$ such that $f(\hat{\mu}(\boldsymbol{s}) \cdot \hat{\mu}(\psi(a)) \cdot e) \neq f(\hat{\mu}(\boldsymbol{s}) \cdot a \cdot e)$. Evidence incompatibility can occur already at symbol initialization or as a result of adding a new evidence due to a counter-example treatment that is described in the sequel. Some elements of $\mu(\boldsymbol{a})$ may behave differently from the representative and this will flag an evidence incompatibility condition to be treated subsequently.

To fix evidence incompatibility, the evidence function μ and the semantics function ψ should be updated such that the total evidence incompatibility degree becomes zero. This is done incrementally and for each state $\boldsymbol{s}$ separately by consecutive calls to EVCOMP until the total incompatibility degree of the observation table becomes zero.

For a state $\boldsymbol{s}$, an incompatibility instance at a indicates either that the partition boundary is imprecise or that a transition (and its corresponding symbol) is missing. Let $\boldsymbol{s} \in \boldsymbol{S}$ be a state; let a be an evidence of the symbol $\boldsymbol{a} \in \boldsymbol{\Sigma_s}$ which is incompatible, i.e., $a \in \mu(\boldsymbol{a})$ and $\text{INC}(\boldsymbol{s}, a) \geq 1$. The algorithm searches for a symbol $\boldsymbol{b} \in \boldsymbol{\Sigma_s}$ for which $f(\hat{\mu}(\boldsymbol{s}) \cdot \hat{\mu}(\boldsymbol{b}) \cdot e) = f(\hat{\mu}(\boldsymbol{s}) \cdot a \cdot e)$. In the favorable case, where such a symbol $\boldsymbol{b}$ exists and belongs to the neighborhood of $\boldsymbol{a}$, the partition

Procedure 12 Make Evidence Compatible

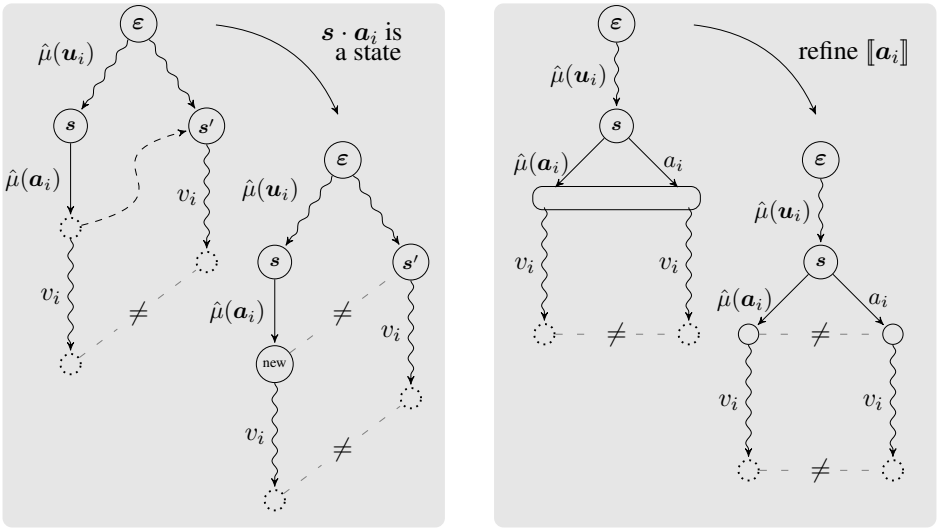
procedure EVCOMP
  Let $\boldsymbol{s} \in \boldsymbol{S}$ be a state with $\mathrm{M}(\boldsymbol{s}) > 0$
  repeat
    Let $a \in \mu_{\boldsymbol{s}}$ such that $\mathrm{INC}(\boldsymbol{s}, a) = 1$ and $\psi_{\boldsymbol{s}}(a) = \boldsymbol{a}$
    if there exists $\boldsymbol{b} \in \boldsymbol{\Sigma_s}$ such that $f_{\hat{\mu}(\boldsymbol{s})\cdot\hat{\mu}(\boldsymbol{b})} = f_{\hat{\mu}(\boldsymbol{s})\cdot\boldsymbol{a}}$ **then**
      $\mu(\boldsymbol{a}) = \mu(\boldsymbol{a}) - \{a\}$; $\mu(\boldsymbol{b}) = \mu(\boldsymbol{b}) \cup \{a\}$
    else
      $\boldsymbol{\Sigma_s} = \boldsymbol{\Sigma_s} \cup \{\boldsymbol{b}\}$ ▷ introduce new symbol $\boldsymbol{b}$
      $\boldsymbol{R} = \boldsymbol{R} \cup \{\boldsymbol{s} \cdot \boldsymbol{b}\}$ ▷ add new boundary element
      $\mu(\boldsymbol{b}) = \{a\}$; $\hat{\mu}(\boldsymbol{b}) = a$
    end if
    Let $\psi(a) = \boldsymbol{b}$
  until $\mathrm{M}(\boldsymbol{s}) = 0$
  Update $\psi_{\boldsymbol{s}}$
end procedure

and boundaries can be modified while preserving convexity of the partition blocks when defining the semantics; the letter a is moved from $\mu(\boldsymbol{a})$ to $\mu(\boldsymbol{b})$. Otherwise, a new symbol $\boldsymbol{b}$ is introduced; the letter a becomes both evidence and representative of $\boldsymbol{b}$ and the incompatibility degree decreases by one. This is repeated for the next incompatible evidence, until $\mathrm{M}(\boldsymbol{s}) = 0$ for all $\boldsymbol{s} \in \boldsymbol{S}$.

Finally, the semantics are updated according to the new evidence function. This may depend on the type of the alphabet but a general approach is to define the semantics according to the closest evidence point. That is $\psi(a) = \boldsymbol{a}$, where $\boldsymbol{a}$ is the symbol whose evidences are closest to a. In other words, we let $\psi(a) = \arg\min_{\boldsymbol{a} \in \boldsymbol{\Sigma_s}} \min_{b \in \mu(\boldsymbol{a})} d(a, b)$, where d denotes some distance measure over the concrete letters.

When the table becomes both closed and evidence compatible we construct from it the hypothesis automaton and pose an equivalence query.

Processing Counter-Examples (Procedure 13). A counter-example is a word w misclassified by the current hypothesis. The conjectured automaton should be modified to classify w correctly while remaining compatible with the evidence accumulated so far. These modifications can be of two major types that we call *vertical* and *horizontal*. The first type, which is the only possible modification in concrete learning, involves the discovery of a new state $\boldsymbol{s} \cdot \boldsymbol{a}$. A counter-example which demonstrates that some letter a took a wrong transition $\boldsymbol{\delta}(\boldsymbol{s}, a)$ has an horizontal effect that fixes a transition or adds a new one. The procedure described in



(a) vertical expansion

(b) horizontal expansion

Figure 5.2: A counter-example expands a hypothesis either (a) vertically, discovering a new state; or (b) horizontally, modifying the alphabet partition in a state.

the sequel reacts to the counter-example by adding a to the evidence of $\boldsymbol{s}$ and thus modifying the table, which should then be made closed and evidence compatible before the learner continues with a new hypothesis.

Counter-examples are treated using a symbolic variant of the breakpoint method, which is described hereafter (for the classical breakpoint method, one can refer to Section 3.3.2). As a matter of fact, a counter-example w may be misclassified also in the new updated hypothesis due to multiple breakpoints. Hence, w can be used again as a counter-example until finding a hypothesis that classifies it correctly.

Let $\boldsymbol{\mathcal{A}_T}$ be a symbolic automaton derived from a symbolic table $\boldsymbol{T}$, and let $w = a_1 \cdots a_{|w|}$ be a counter-example whose symbolic image is $\boldsymbol{a}_1 \cdots \boldsymbol{a}_{|w|}$. An *$i$-factorization* of w is $w = u_i \cdot a_i \cdot v_i$ such that $u_i = a_1 \cdots a_{i-1}$ and $v_i = a_{i+1} \cdots a_{|w|}$. For every i-factorization of w, let $\boldsymbol{u}_i$ be the symbolic image of u_i, and $\boldsymbol{s_i} = \boldsymbol{\delta}(\varepsilon, \boldsymbol{u}_i \cdot \boldsymbol{a}_i)$ be the symbolic state (an element of $\boldsymbol{S}$) reached in $\boldsymbol{\mathcal{A}_T}$ after reading $\boldsymbol{u}_i \cdot \boldsymbol{a}_i$.

Proposition 5.6 (Symbolic Breakpoint)**.** *If w is a counter-example to $\boldsymbol{\mathcal{A}_T}$ then there exists an i-factorization of w such that either*

$$f(\hat{\mu}(\boldsymbol{s}_{i-1}) \cdot a_i \cdot v_i) \neq f(\hat{\mu}(\boldsymbol{s}_{i-1}) \cdot \hat{\mu}(\boldsymbol{a}_i) \cdot v_i) \tag{5.2}$$

or

$$f(\hat{\mu}(\boldsymbol{s}_{i-1} \cdot \boldsymbol{a}_i) \cdot v_i) \neq f(\hat{\mu}(\boldsymbol{s}_i) \cdot v_i) \tag{5.3}$$

Proof. Condition (5.2) states that a_i is not well represented by $\hat{\mu}(\boldsymbol{a}_i)$ while Condition (5.3) implies $\boldsymbol{s}_{i-1} \cdot \boldsymbol{a}_i$ is a new state different from $\boldsymbol{s}_i$, see Figure 5.2. We prove the proposition assuming that none of the inequalities above hold for any i-factorization of w. Then, by alternatively using the negations of (5.2) and (5.3) for all values of i, we conclude that $f(\hat{\mu}(\boldsymbol{s}_0)\cdot a_1 \cdot v_1) = f(\hat{\mu}(\boldsymbol{s}_{|w|}))$, where $\hat{\mu}(\boldsymbol{s}_0)\cdot a_1 \cdot v_1$ is the counter-example and $\boldsymbol{s}_{|w|}$ is the state reached in $\boldsymbol{\mathcal{A}_T}$ after reading w. Hence, w is not a counter-example, which is not true.

This argument is illustrated in more detail below. The actual counter-example $w = a_1 \dots a_m$ is shown as an upper-script, such that the factorization of w is clear at each step.

$$
\begin{aligned}
f(w) = \; & f(\hat{\mu}(\boldsymbol{s}_0)^{\varepsilon} \cdot a_1^{a_1} \cdot v_1^{a_2 \dots a_m}) && \overset{(5.2)}{=} f(\hat{\mu}(\boldsymbol{s}_0)^{\varepsilon} \cdot \hat{\mu}(\boldsymbol{a}_1)^{a_1} \cdot v_1^{a_2 \dots a_m}) && \overset{(5.3)}{=} \\
= \; & f(\hat{\mu}(\boldsymbol{s}_1)^{a_1} \cdot a_2^{a_2} \cdot v_2^{a_3 \dots a_m}) && \overset{(5.2)}{=} f(\hat{\mu}(\boldsymbol{s}_1)^{a_1} \cdot \hat{\mu}(\boldsymbol{a}_2)^{a_2} \cdot v_2^{a_3 \dots a_m}) && \overset{(5.3)}{=} \\
& \vdots \\
= \; & f(\hat{\mu}(\boldsymbol{s}_{i-1})^{a_1 \dots a_{i-1}} \cdot a_i^{a_i} \cdot v_i^{a_{i+1} \dots a_m}) && \overset{(5.2)}{=} f(\hat{\mu}(\boldsymbol{s}_{i-1})^{a_1 \dots a_{i-1}} \cdot \hat{\mu}(\boldsymbol{a}_i)^{a_i} \cdot v_i^{a_{i+1} \dots a_m}) && \overset{(5.3)}{=} \\
& \vdots \\
= \; & f(\hat{\mu}(\boldsymbol{s}_{m-1})^{a_1 \dots a_{m-1}} \cdot a_m^{a_m} \cdot v_m^{\varepsilon}) && \overset{(5.2)}{=} f(\hat{\mu}(\boldsymbol{s}_{m-1})^{a_1 \dots a_{m-1}} \cdot \hat{\mu}(\boldsymbol{a}_m)^{a_m} \cdot v_m^{\varepsilon}) && \overset{(5.3)}{=} \\
= \; & f(\hat{\mu}(\boldsymbol{s}_m)^{a_0 \dots a_m}) &&
\end{aligned}
$$

□

Let us consider a hypothesis automaton $\mathcal{A}_{\boldsymbol{T}}$ for which $w \in \Sigma^*$ is given as a counter-example. We use the symbolic breakpoint method to treat this counter-example and update the symbolic table accordingly. As we can see in Procedure 13, the learner iterates over i values and checks whether one of the conditions (5.2) and (5.3) holds for some i-factorization of the counter-example. We can choose i in any order, as described in Section 3.3.2, either to retain smaller prefixes or suffixes, or to find a breakpoint faster. We let i take values in a monotonically descending order and keep the suffixes as short as possible. In this case, it suffices to compare $f(\hat{\mu}(\boldsymbol{s}_{i-1} \cdot \boldsymbol{a}_i) \cdot v_i)$ and $f(\hat{\mu}(\boldsymbol{s}_{i-1}) \cdot a_i \cdot v_i)$ with the classification of the counter-example, which is kept in a flag variable. When we iterate (line 3) in a descending order, the flag value (line 2) should be set to flag $= f(w)$, that is, the classification of the counter-example by the Teacher.

The two main conditions of the breakpoint method are checked in lines 5 and 9. Specifically, the learner checks first condition (5.3) in line 5, where its satisfaction implies that a new distinguishing string has been found. Adding v_i to E will distinguish between states $\boldsymbol{s}_{i-1} \cdot \boldsymbol{a}_i$ and $\boldsymbol{s}_i$, resulting a non closed table. Otherwise, if condition (5.2) holds, a check made in line 9, it means that the letter a_i does not

Procedure 13 Counter-Example Treatment

1: **procedure** COUNTEREX($\mathcal{A}_{\boldsymbol{T}}, w$)
2: flag $= f(\hat{\mu}(\boldsymbol{\delta}(\boldsymbol{\varepsilon}, \boldsymbol{w})))$ ▷ flag $= f(w)$ when iterating over $1, \ldots, |w|$
3: **for** $i = |w|, \ldots, 1$ **do**
4: For an i-factorization $w = u_i \cdot a_i \cdot v_i$
5: **if** $f(\hat{\mu}(\boldsymbol{s}_{i-1} \cdot \boldsymbol{a}_i) \cdot v_i) \neq$ flag **then** ▷ check (5.3)
6: $E = E \cup \{v_i\}$ ▷ add v_i as a new distinguishing word
7: Ask MQ(u) for all $u \in \mu(\boldsymbol{S} \cup \boldsymbol{R}) \cdot v_i$
8: **break**
9: **else if** $f(\hat{\mu}(\boldsymbol{s}_{i-1}) \cdot a_i \cdot v_i) \neq$ flag **then** ▷ check (5.2)
10: $\mu(\boldsymbol{a}_i) = \mu(\boldsymbol{a}_i) \cup \{a_i\}$ ▷ add a_i as new evidence
11: Ask MQ(u) for all $u \in \hat{\mu}(\boldsymbol{s}_{i-1}) \cdot a_i \cdot E$
12: **if** M$(\boldsymbol{s}_i) = 0$ **then**
13: $E = E \cup \{v_i\}$ ▷ add distinguishing word when needed
14: Ask MQ(u) for all $u \in \mu(\boldsymbol{S} \cup \boldsymbol{R}) \cdot v_i$
15: **end if**
16: **break**
17: **end if**
18: **end for**
19: **end procedure**

behave as its representative in the hypothesis. As a result, the letter a_i is added to the evidence of $\boldsymbol{a}_i$ and new membership queries are posed to fill in the table. These queries will render the table evidence incompatible and will lead to refining of $[\![\boldsymbol{a}_i]\!]$. If no evidence incompatibility occurs after introducing the new evidence and filling in the table, this is due to a missing distinguishing string, which is v_i that is not yet part of the table. In such a case, also the suffix v_i is added to E, as it is the only witness for the incompatibility, see lines 12-15. The procedure terminates when either condition (5.2) or condition (5.3) is satisfied (line 9 and line 5).

Note here that checking the conditions involves supplementary membership queries that are based on the suffix of the counter-example w where the prefix u_i of w is replaced by $\hat{\mu}(\boldsymbol{s}_{i-1})$, the representative of its shortest equivalent symbolic word in the table. Both cases will lead to a new conjectured automaton that might still not classify w correctly. Hence, if w remains a counter-example for the new conjecture, Procedure 13 should be invoked with the same counter-example until the new hypothesis $\mathcal{A}_{\boldsymbol{T}}$ classifies w correctly.

Chapter 6
Learning with a Helpful Teacher

In this chapter, we present an instantiation of the learning algorithm that uses the strong assumption of a helpful teacher. Such a teacher responds positively to an equivalence query $\text{EQ}(\mathcal{A})$, where $\mathcal{A}$ is an automaton conjectured by the learning algorithm, only if $L(\mathcal{A})$ is indeed equivalent to the target language; otherwise, it returns a *minimal counter-example* which is a counter-example of minimal length which is also minimal with respect to a lexicographic order. A *minimal counter-example* helps the learner to localize the modification site. This strong assumption improves the performance of the algorithm resulting in an exact conjecture; its relaxation is discussed in Chapter 7. In the following sections, we discuss the learning problem for the cases of totally and partially-ordered input alphabets, respectively.

6.1 Learning Languages over Ordered Alphabets

In this section, we assume Σ to be a *totally-ordered* alphabet with minimum and restrict ourselves to interval symbolic automata (see Section 4.4.1). In these automata the concrete alphabet is any left-closed subset of $\mathbb{N}, \mathbb{Q}$ or $\mathbb{R}$, and the semantics, in the case of a dense order as in $\mathbb{R}$, form closed-open intervals.

The order on the alphabet is extended naturally to a lexicographic order on Σ^*. The minimality of the counter-example, in length and lexicographically, improves the learning procedure. The treatment of counter-examples becomes cleaner and learning the semantics is accurate. Minimality allows us to use fewer evidences, keep the concrete sample as small as possible, and avoid posing unnecessary membership queries.

In Section 5.3 we have seen that new symbolic letters are introduced on two occasions: when a new state is discovered or when a partition is modified due to

a counter-example. In both cases, we set the concrete semantics $[\![\boldsymbol{a}]\!]$ to the largest possible subset of Σ, given the current evidence.

The assumption of a minimal counter-example allows us to specialize some of the procedures of Chapter 5, which results in fewer evidences and a cleaner and cheaper counter-examples treatment.

For each symbol $\boldsymbol{a}$ in $\boldsymbol{\Sigma}$ we choose exactly one letter as evidence, that is the smallest possible $a \in [\![\boldsymbol{a}]\!]$. As this minimal element is the only evidence, we choose it also as the representative. Let us denote by a_0 the minimal element of the input alphabet, $a_0 = \min \Sigma$. We adapt Procedure 10, the initialization of a new symbol, to this setting. We replace $sample(\Sigma, k)$ by $\{a_0\}$; the evidence, representative, and semantics functions are updated such that, for each symbol $\boldsymbol{a} \in \boldsymbol{\Sigma}$, the following holds:

$$\mu(\boldsymbol{a}) = \{\hat{\mu}(\boldsymbol{a})\} = \{\min[\![\boldsymbol{a}]\!]\}. \tag{6.1}$$

A misclassified word in the conjectured automaton is due to a wrong transition that has been taken. This happens in two cases: either some evidence is missing, or the target state has not been discovered yet. Unlike the general case described in Section 5.3, minimality assures that the counter-example uses the representatives, which are the smallest elements of each transition, to reach states in the automaton through a loop-free path. After detecting the state at which the hypothesis is problematic, the missing state or a transition is added appropriately. Let w be a counter-example, since this is minimal it admits an i-factorization $w = u_i \cdot a_i \cdot v_i$, with u_i being the largest prefix of w such that $u_i \in \mu(\boldsymbol{u}_i)$ for some $\boldsymbol{u}_i \in \boldsymbol{S} \cup \boldsymbol{R}$ but $u_i \cdot a_i \notin \mu(\boldsymbol{u}')$ for any prefix $\boldsymbol{u}'$ in $\boldsymbol{S} \cup \boldsymbol{R}$. We consider two cases, $\boldsymbol{u}_i \in \boldsymbol{S}$ and $\boldsymbol{u}_i \in \boldsymbol{R}$.

In the first case, when $\boldsymbol{u}_i$ is already a state in the hypothesis, i.e., $\boldsymbol{u}_i \in \boldsymbol{S}$, the letter a_i indicates that the partition boundaries are not correctly defined and need refinement. Letter a_i is wrongly considered to be part of $[\![\boldsymbol{a}_i]\!]$; a new symbol $\boldsymbol{b}$ is added to $\boldsymbol{\Sigma}_{\boldsymbol{u}_i}$, with $\hat{\mu}(\boldsymbol{b}) = a_i$, and solves the evidence incompatibility. Due to minimality of the counter-example, all letters in $[\![\boldsymbol{a}_i]\!]$ less than letter a_i behave like $\hat{\mu}(\boldsymbol{a}_i)$. We assume that all remaining letters in $[\![\boldsymbol{a}_i]\!]$ behave like a_i and map them to $\boldsymbol{b}$. The semantics function $\psi_{\boldsymbol{u}_i}$ is updated such that $\psi_{\boldsymbol{u}_i}(a) = \boldsymbol{b}$ for all $a \in [\![\boldsymbol{a}_i]\!], a \geq a_i$, and $\psi_{\boldsymbol{u}_i}(a)$ stays unchanged otherwise. Finally, the prefix $\boldsymbol{u}_i \cdot \boldsymbol{b}$ is added to $\boldsymbol{R}$.

In the second case, when the symbolic word $\boldsymbol{u}_i$ is part of the boundary, $\boldsymbol{u}_i \in \boldsymbol{R}$; we infer that $\boldsymbol{u}_i$ is not equivalent to any of the existing states in the hypothesis. Let $\boldsymbol{s} \in \boldsymbol{S}$ be the prefix that was considered to be equivalent to $\boldsymbol{u}_i$ for which $\boldsymbol{f}_{\boldsymbol{u}_i} = \boldsymbol{f}_{\boldsymbol{s}}$. Since the table is reduced $\boldsymbol{s}$ is unique. Because w is minimal, the classification of words in $[\![\boldsymbol{s}]\!] \cdot a_i \cdot v_i$ in the hypothesis automaton is correct; otherwise, there exists some $s \in [\![\boldsymbol{s}]\!]$ such that $s \cdot a_i \cdot v_i$ constitutes a shorter counter-example. From this we

Procedure 14 Counter-Example Treatment (with Helpful Teacher) - $\mathbb{R}$

procedure COUNTEREX(w)
  Find an i-factorization $w = u_i \cdot a_i \cdot v_i$, $a_i \in \Sigma$, u_i, $v_i \in \Sigma^*$ such that
    $\exists \boldsymbol{u_i} \in \boldsymbol{S} \cup \boldsymbol{R}$, $u_i \in \mu(\boldsymbol{u_i})$ and $u_i \cdot a_i \notin \mu(\boldsymbol{u'}), \forall \boldsymbol{u'} \in \boldsymbol{S} \cup \boldsymbol{R}$
  if $\boldsymbol{u_i} \in \boldsymbol{S}$ **then**
    Let $\boldsymbol{a_i} \in \boldsymbol{\Sigma_{u_i}}$ such that $a_i \in [\![\boldsymbol{a_i}]\!]$ ▷ refinement of $[\![\boldsymbol{a_i}]\!]$
    $\boldsymbol{\Sigma_{u_i}} = \boldsymbol{\Sigma_{u_i}} \cup \{\boldsymbol{b}\}$ ▷ introduce new symbol $\boldsymbol{b}$
    $\mu(\boldsymbol{b}) = \{a_i\}$; $\hat{\mu}(\boldsymbol{b}) = a_i$
    $[\![\boldsymbol{b}]\!] = \{a \in [\![\boldsymbol{a_i}]\!] : a \geq a_i\}$
    $[\![\boldsymbol{a_i}]\!] = [\![\boldsymbol{a_i}]\!] \setminus [\![\boldsymbol{b}]\!]$
    $\boldsymbol{R} = \boldsymbol{R} \cup \{\boldsymbol{u_i} \cdot \boldsymbol{b}\}$
    Ask MQ(u) for all $u \in \mu(\boldsymbol{u_i} \cdot \boldsymbol{b}) \cdot E$
  else ▷ if $\boldsymbol{u_i} \in \boldsymbol{R}$
    $\boldsymbol{S} = \boldsymbol{S} \cup \{\boldsymbol{u_i}\}$; $E = E \cup \{a_i \cdot v_i\}$ ▷ $\boldsymbol{u_i}$ becomes a new state
    if $a_i = a_0$ **then**
      $\boldsymbol{\Sigma_{u_i}} = \{\boldsymbol{b}\}$
      $\mu(\boldsymbol{b}) = \{a_0\}$; $\hat{\mu}(\boldsymbol{b}) = a_0$; $[\![\boldsymbol{b}]\!] = \Sigma$
    else
      $\boldsymbol{\Sigma_{u_i}} = \{\boldsymbol{b}, \boldsymbol{b'}\}$
      $\mu(\boldsymbol{b}) = \{a_0\}$; $\hat{\mu}(\boldsymbol{b}) = a_0$; $[\![\boldsymbol{b}]\!] = \{a \in \Sigma : a < a_i\}$
      $\mu(\boldsymbol{b'}) = \{a_i\}$; $\hat{\mu}(\boldsymbol{b'}) = a_i$; $[\![\boldsymbol{b'}]\!] = \{a \in \Sigma : a \geq a_i\}$
    end if
    $\boldsymbol{R} = (\boldsymbol{R} - \{\boldsymbol{u_i}\}) \cup \boldsymbol{u_i} \cdot \boldsymbol{\Sigma_{u_i}}$
    Ask MQ(u) for all $u \in \mu(\boldsymbol{u_i} \cdot \boldsymbol{\Sigma_{u_i}}) \cdot E$
  end if
end procedure

conclude that $\boldsymbol{u_i}$ is a new state and the hypothesis automaton should be expanded vertically moving $\boldsymbol{u_i}$ to $\boldsymbol{S}$. To distinguish between $\boldsymbol{u_i}$ and $\boldsymbol{s}$, the suffix $a_i \cdot v_i$ is added to E. The letter a_i is an evidence for the new state $\boldsymbol{u_i}$ and, for 5.1 to hold, we distinguish the following two sub-cases. If a_i is the smallest element of Σ, i.e., $a_i = a_0$, one symbolic letter $\boldsymbol{b}$ is initialized and added to $\boldsymbol{\Sigma_{u_i}}$. Otherwise, if a_i is not the smallest and $a_i > a_0$, *two* new symbolic letters, $\boldsymbol{b}$ and $\boldsymbol{b'}$, are added to $\boldsymbol{\Sigma_{u_i}}$ with $[\![\boldsymbol{b}]\!] = \{a : a < a_i\}$, $[\![\boldsymbol{b'}]\!] = \{a : a \geq a_i\}$, $\hat{\mu}(\boldsymbol{b}) = a_0$, and $\hat{\mu}(\boldsymbol{b'}) = a_i$. The prefixes $\boldsymbol{u} \cdot \boldsymbol{\Sigma_{u_i}}$ are added to $\boldsymbol{R}$.

The counter-example treatment for ordered alphabets and minimal counter-examples is summarized in Procedure 14. This procedure significantly differs from the general Procedure 13 discussed in the previous section. More precisely, evidence incompatibility appears only during the counter-example treatment where it is instantly solved. Hence, when the Procedure 13 is replaced by Procedure 14,

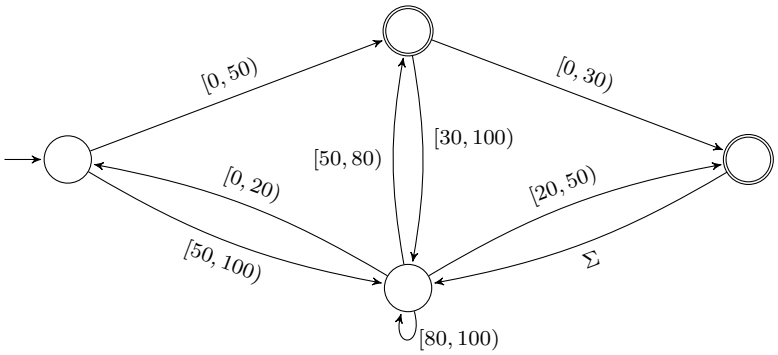


Figure 6.1: Minimal automaton accepting the target language used in Example 6.1.

then EVCOMP in Algorithm 8 is never invoked.

However, one can use Procedure 13 instead of Procedure 14, with one minor modification in line 2, where the iteration order should be changed to increasing, and the flag should be initialized to $f(w)$. In such a case, the procedure EVCOMP should be called whenever necessary. Although the final conjecture will be the same, the algorithm becomes less efficient in the sense that more repetitions in the main algorithm are needed to reach the same result. Moreover, the algorithm calls COUNTEREX (Procedure 13) more times, and a number of additional MQ's are posed in order to perform the tests in lines 5 and 9. Note that MQ's are not required at all when the counter-example is treated using Procedure 14.

Example 6.1 (Learning Interval Symbolic Automata with Minimal Counter-Examples)**.** Let the concrete alphabet be $\Sigma = [0, 100) \subset \mathbb{R}$ equipped with the usual order, and let $L \subseteq \Sigma^*$ be a target language whose automaton is shown in Figure 6.1. Figure 6.2 shows the evolution of the symbolic observation tables and Figure 6.3 depicts the corresponding automata and the concrete semantics of the symbolic alphabets after each update.

The learner initializes the table by letting $\boldsymbol{S} = \{\boldsymbol{\varepsilon}\}$, $\boldsymbol{\Sigma} = \{\boldsymbol{a}_0\}$, $\boldsymbol{R} = \{\boldsymbol{a}_0\}$, $\mu(\boldsymbol{a}_0) = \{0\}$, and $E = \{\varepsilon\}$. The initial observation table $\boldsymbol{T}_0$ is filled in after membership queries are posed for ε, which is rejected, and for 0, which is accepted. To close the table, the learner makes $\boldsymbol{a}_0$ a new state and introduces $\boldsymbol{\Sigma}_{\boldsymbol{a}_0} = \{\boldsymbol{a}_1\}$, where $\boldsymbol{a}_1$ is a new symbol with $\mu(\boldsymbol{a}_1) = \{0\}$. The symbolic word $\boldsymbol{a}_0 \cdot \boldsymbol{a}_1$ is then added to $\boldsymbol{R}$. A membership query $\text{MQ}(0 \cdot 0)$ fills the table. The new observation table is $\boldsymbol{T}_1$ and it is closed. The symbolic automaton $\boldsymbol{\mathcal{A}}_1$ derived from $\boldsymbol{T}_1$ has two states, one accepting and one rejecting.

The learner poses an equivalence query for $\boldsymbol{\mathcal{A}}_1$ and obtains $(50, -)$ as a minimal counter-example. The counter-example's minimality implies that all one-letter words smaller than 50 are correctly classified, in contrast to words greater than

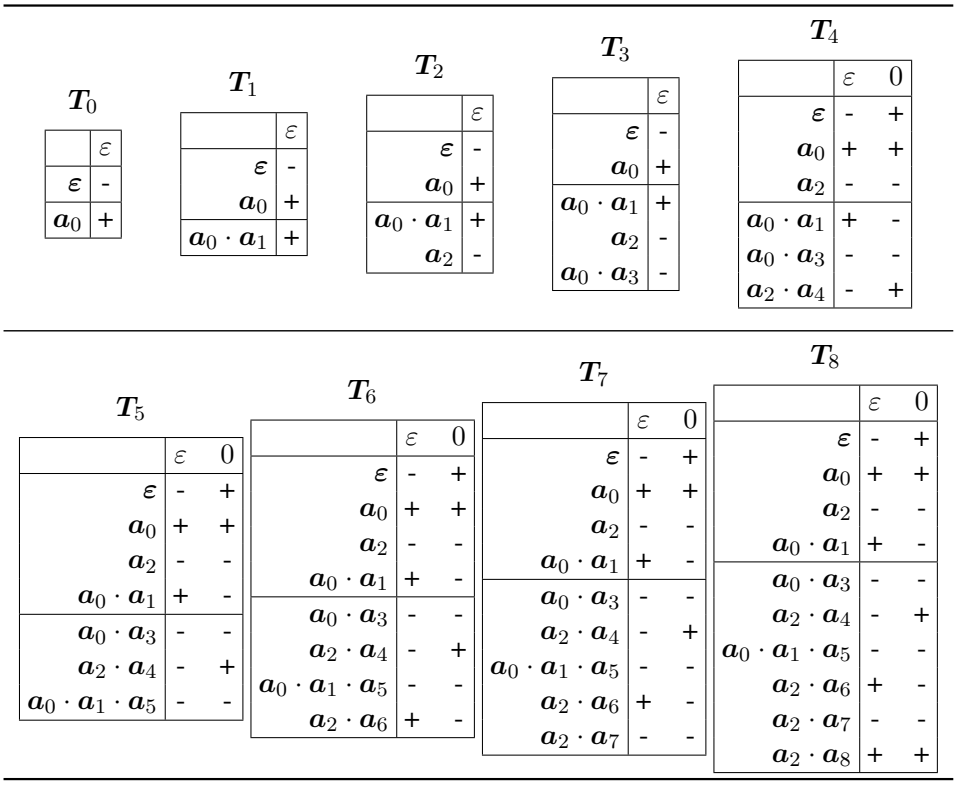


Figure 6.2: Observation tables used in Example 6.1.

50. For this reason, the symbol $\boldsymbol{a}_0$ is not a good representation for all Σ, and a new symbol $\boldsymbol{a}_2$ is added to $\boldsymbol{\Sigma}_{\boldsymbol{\varepsilon}}$. The concrete semantics for this state is updated to $[\![\boldsymbol{a}_0]\!] = \{a \in \Sigma : a < 50\}$ and $[\![\boldsymbol{a}_2]\!] = \{a \in \Sigma : a \geq 50\}$. As an evidence the smallest possible letter is selected, i.e., $\mu(\boldsymbol{a}_2) = \{50\}$. The learner asks membership queries, fills in the table, which is closed and appears in Figure 6.2 as $\boldsymbol{T}_2$, and constructs the new hypothesis $\boldsymbol{\mathcal{A}}_2$.

As the symbolic automaton $\boldsymbol{\mathcal{A}}_2$ is not the correct one, the teacher returns the counter-example $(0 \cdot 30, -)$. The prefix 0 already exists in the sample, which means that the misclassification must occur in the second transition, that is, at $\boldsymbol{a}_1$. Therefore, the alphabet partition for state $\boldsymbol{a}_0$ is refined by introducing a new symbol $\boldsymbol{a}_3$ and by letting $[\![\boldsymbol{a}_1]\!] = \{a \in \Sigma : a < 30\}$ and $[\![\boldsymbol{a}_3]\!] = \{a \in \Sigma : a \geq 30\}$. The new table $\boldsymbol{T}_3$ is closed, and the new hypothesis automaton is $\boldsymbol{\mathcal{A}}_3$. Again, this hypothesis automaton does not recognize the target language, and $(50 \cdot 0, -)$ is a new minimal counter-example. The prefix 50 belongs to the evidence of $\boldsymbol{a}_2$; this signifies that $\boldsymbol{a}_2$ goes to a state that is different from $\boldsymbol{\varepsilon}$. The suffix 0 of the counter-example is added to E to distinguish the two states. The learner moves $\boldsymbol{a}_2$ to the set of states and introduces the set $\boldsymbol{\Sigma}_{\boldsymbol{a}_2} = \{\boldsymbol{a}_4\}$, where $\boldsymbol{a}_4$ is a new symbol with

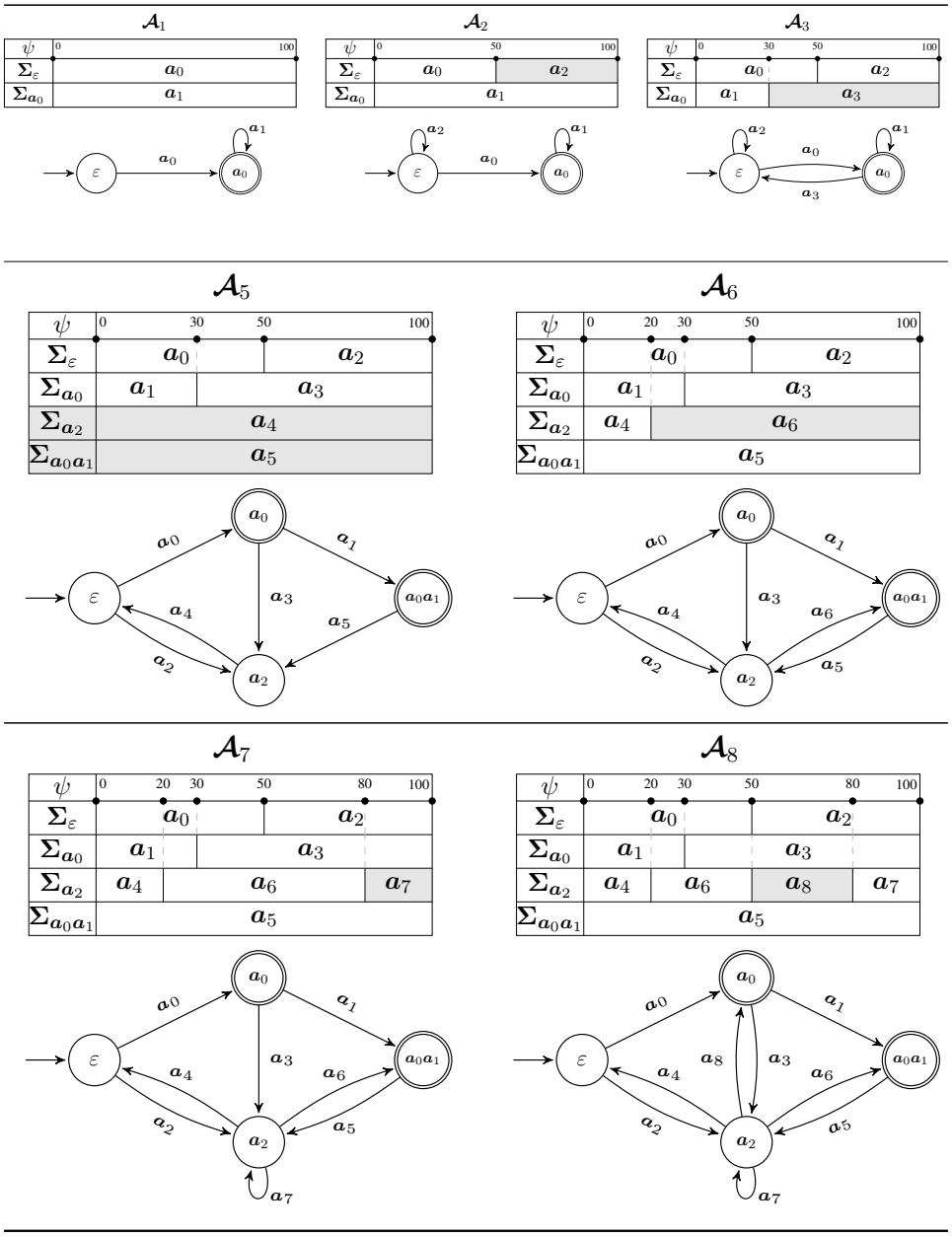


Figure 6.3: Hypotheses and Σ-semantics as learned in Example 6.1.

$\mu(\boldsymbol{a}_4) = \{0\}$. Then, $\boldsymbol{a}_2 \cdot \boldsymbol{a}_4$ is added to $\boldsymbol{R}$. After all updates, the new table $\boldsymbol{T}_4$ is made closed, resulting in $\boldsymbol{T}_5$ and the conjectured automaton $\boldsymbol{A}_5$, which has two new states.

In the following, three more counter-examples $(50 \cdot 20, +)$, $(50 \cdot 80, -)$ and $(50 \cdot 50 \cdot 0, +)$ are provided. Progressively, each of them expands horizontally

the hypothesis by adding a new transition. The automata, after each update, are $\mathcal{A}_6$, $\mathcal{A}_7$ and $\mathcal{A}_8$, respectively. The learner terminates with the last automaton $\mathcal{A}_8$ accepting the target language.

At this point, it is important to note that the language L that is defined over a subset of $\mathbb{R}$ cannot be learned by the classical L^* algorithm. To be able to compare it with this algorithm, we restrict the problem to a finite discrete alphabet, such that $\Sigma = \{1, \ldots, 100\}$, and let the target language be recognized from the same symbolic automaton. The L^* algorithm requires approximately 1000 queries to learn L, in contrast to a total of 17 queries that are required by the symbolic algorithm. □

6.2 Learning over Partially-ordered Alphabets

Let Σ be a partially-ordered alphabet such as a bounded sub-rectangle of $\mathbb{R}^n$, i.e., $\Sigma = X^d$, where X is a totally-ordered set such as a sub-interval $[0, k) \subseteq \mathbb{R}$. The target language L is accepted by a canonical symbolic automaton where partitions are monotone, as described in Section 4.4.2. Monotone partitions of this kind are defined using finite unions of forward cones, and equivalently, a partition block can be determined by a finite set of minimum mutually incomparable letters from Σ. We denote a monotone partition by $\mathcal{P} = \{P_1, \ldots, P_{m-1}\}$, where $P_i = B^+(F_{i-1}) - B^+(F_i)$ and $B^+(F)$ is a forward cone defined by a set of incomparable points $F \subseteq \Sigma$.

The partial order on the alphabet extends to a partial lexicographic order over Σ^*. The teacher is assumed to return a minimal counter-example, which in this case is chosen from a finite set of incomparable minimal counter-examples. As for totally-ordered alphabets, the minimality of the counter-examples improves the learning algorithm, resulting in fewer membership queries for counter-example treatment, and to a precise final hypothesis. Evidences derived from counter-examples denote the minimal points of a partition block. Hence, multiple evidences are used to determine each partition block. In the following, we present the symbolic learning algorithm adapted to this special nature of the partitions. We mainly describe the modifications that should be considered to handle the evidences and solve incompatibility whenever this occurs.

The main algorithm (Algorithm 8) is applied and the procedure of table initialization, symbol initializations, and table closure (see Procedures 9, 10 and 11) remain unchanged.

Counter-examples either discover new states or refine partitions. When refinement occurs, minimality of the counter-examples constrains the teacher to provide all the counter-examples needed to find the minimal points that completely define

Procedure 15 Counter-Example Treatment (with Helpful Teacher) - $\mathbb{R}^n$

procedure COUNTEREX(w)
  Find a i-factorization $w = u_i \cdot a_i \cdot v_i$, $a_i \in \Sigma$, u_i, $v_i \in \Sigma^*$ such that
    $\exists \boldsymbol{u}_i \in \boldsymbol{S} \cup \boldsymbol{R}$, $u_i \in \mu(\boldsymbol{u}_i)$ and $u_i \cdot a_i \notin \mu(\boldsymbol{u'}), \forall \boldsymbol{u'} \in \boldsymbol{S} \cup \boldsymbol{R}$
  if $\boldsymbol{u}_i \in \boldsymbol{S}$ **then**
    Let $\boldsymbol{a}_i \in \boldsymbol{\Sigma}_{\boldsymbol{u}_i}$ such that $a_i \in [\![\boldsymbol{a}_i]\!]$
    Let $\boldsymbol{a} \in \boldsymbol{\Sigma}_{\boldsymbol{u}_i}$ such that $\boldsymbol{a} > \boldsymbol{a}_i$ adjacent to $\boldsymbol{a}_i$ ▷ $\boldsymbol{a} \in \mathcal{V}(\boldsymbol{a}_i)$
    if $a_i || a$ for all $a \in \mu(\boldsymbol{a})$ and $f_{u_i \cdot a_i} = f_{u_i \cdot \hat{\mu}(\boldsymbol{a})}$ **then**
      $\mu(\boldsymbol{a}) = \mu(\boldsymbol{a}) \cup \{a_i\}$ ▷ change boundary
      $[\![\boldsymbol{a}]\!] = [\![\boldsymbol{a}]\!] \cup \{a \in [\![\boldsymbol{a}_i]\!] : a \geq a_i\}$
      $[\![\boldsymbol{a}_i]\!] = [\![\boldsymbol{a}_i]\!] \setminus \{a \in \Sigma : a \geq a_i\}$
    else ▷ partition refinement
      $\boldsymbol{\Sigma}_{\boldsymbol{u}_i} = \boldsymbol{\Sigma}_{\boldsymbol{u}_i} \cup \{\boldsymbol{b}\}$ ▷ introduce new symbol $\boldsymbol{b}$
      $\mu(\boldsymbol{b}) = \{a_i\}$; $\hat{\mu}(\boldsymbol{b}) = a_i$
      $[\![\boldsymbol{b}]\!] = \{a \in [\![\boldsymbol{a}_i]\!] : a \geq a_i\}$
      $[\![\boldsymbol{a}_i]\!] = [\![\boldsymbol{a}_i]\!] \setminus [\![\boldsymbol{b}]\!]$
      $\boldsymbol{R} = \boldsymbol{R} \cup \{\boldsymbol{u}_i \cdot \boldsymbol{b}\}$
      Ask MQ(u) for all $u \in \mu(\boldsymbol{u}_i \cdot \boldsymbol{b}) \cdot E$
    end if
  else ▷ if $\boldsymbol{u}_i \in \boldsymbol{R}$
    $\boldsymbol{S} = \boldsymbol{S} \cup \{\boldsymbol{u}_i\}$; $E = E \cup \{a_i \cdot v_i\}$ ▷ $\boldsymbol{u}_i$ becomes a new state
    if $a_i = a_0$ **then**
      $\boldsymbol{\Sigma}_{\boldsymbol{u}_i} = \{\boldsymbol{b}\}$ ▷ introduce new symbol $\boldsymbol{b}$
      $\mu(\boldsymbol{b}) = \{a_0\}$; $\hat{\mu}(\boldsymbol{b}) = a_0$; $[\![\boldsymbol{b}]\!] = \Sigma$
    else
      $\boldsymbol{\Sigma}_{\boldsymbol{u}_i} = \{\boldsymbol{b}, \boldsymbol{b}'\}$ ▷ introduce new symbols $\boldsymbol{b}$ and $\boldsymbol{b}'$
      $\mu(\boldsymbol{b}) = \{a_i\}$; $\hat{\mu}(\boldsymbol{b}) = a_i$; $[\![\boldsymbol{b}]\!] = \{a \in \Sigma : a \geq a_i\}$
      $\mu(\boldsymbol{b}') = \{a_0\}$; $\hat{\mu}(\boldsymbol{b}') = a_0$; $[\![\boldsymbol{b}']\!] = \Sigma \setminus [\![\boldsymbol{b}]\!]$
    end if
    $\boldsymbol{R} = (\boldsymbol{R} - \{\boldsymbol{u}_i\}) \cup \boldsymbol{u}_i \cdot \boldsymbol{\Sigma}_{\boldsymbol{u}_i}$
    Ask MQ(u) for all $u \in \mu(\boldsymbol{u}_i \cdot \boldsymbol{\Sigma}_{\boldsymbol{u}_i}) \cdot E$
  end if
end procedure

the new partition. Even though the counter-examples are provided in an arbitrary order, defined by the teacher, this does not affect the behavior of the algorithm.

Let $\boldsymbol{s} \in \boldsymbol{S}$ be a state in the observation table T, and let $\boldsymbol{a}$ be a symbol in $\boldsymbol{\Sigma}_{\boldsymbol{s}}$. The set of evidences of $\boldsymbol{a}$ is the set of the mutually-incomparable minimum elements belonging to its semantics. These evidences are sufficient to define each partition block as the set difference between two unions of cones, each cone defined by a letter $a \in \mu(\boldsymbol{a})$. When $\boldsymbol{s}$ is a new state, we let $\boldsymbol{\Sigma}_{\boldsymbol{s}} = \{\boldsymbol{a}\}$, where $\boldsymbol{a}$ is a new

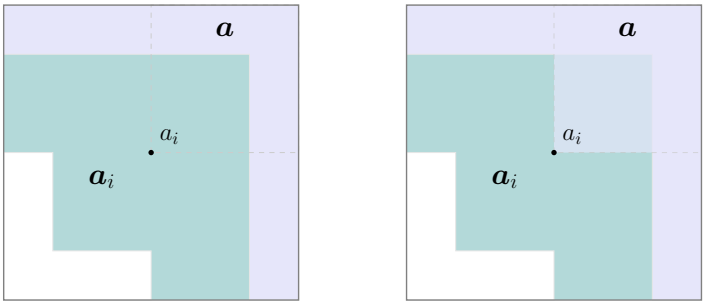


Figure 6.4: Modifying the alphabet partition for state $\boldsymbol{u}_i$ after receiving $u_i \cdot a_i \cdot v_i$ as counter-example. Letters greater than a_i are moved from $[\![\boldsymbol{a}_i]\!]$ to $[\![\boldsymbol{a}]\!]$.

symbol which is initialized; the concrete semantics $[\![\boldsymbol{a}]\!]$ is set to Σ and the minimal element $a_0 = \mathbf{0}$ serves as evidence and representative. Evidences are added later, given by counter-examples. New evidences refine the partition either by adding a new symbol, or by modifying the set of cones for an existing partition block. Symbols may have many evidences, but keep as representative the first evidence admitted.

Procedure 14, that treats the counter-examples for ordered alphabets, should be modified in the case where the counter-example only modifies a partition boundary, without adding new states or transitions. For this an additional condition should be checked. The prefix u_i belongs to a state, but the new letter $a_i \in [\![\boldsymbol{a}_i]\!]$, behaves like the adjacent symbol $\boldsymbol{a} > \boldsymbol{a}_i$. There is no need of a new symbol and the evidence incompatibility is solved by moving a_i from $[\![\boldsymbol{a}_i]\!]$ to $[\![\boldsymbol{a}]\!]$ along with all letters greater or equal to it. The letter a_i is then added to the evidences of $\boldsymbol{a}$. Figure 6.4 shows this update of the semantics. The full modification of the procedure appears in Procedure 15. In the lines 8-10, the learning handles the case of boundary modification. The rest of the procedure remains unchanged, and one or two new symbols are introduced to refine the existing partition block that is evidence incompatible.

Example 6.2. Let us illustrate the working of the algorithm on a target language L defined over $\Sigma = [0, 100]^2$. We assume partitions to be monotone. All resulting tables, hypotheses automata and alphabet partitions for this example are shown in Figures 6.5, 6.6, and 6.7, respectively.

The learner starts by asking a membership query about the empty word. A symbolic letter $\boldsymbol{a}_0$ is chosen to represent the continuations from the initial state, initially placed in the boundary. The learner chooses the minimal element of Σ as evidence, i.e., $\boldsymbol{\Sigma}_\varepsilon = \{\boldsymbol{a}_0\}$, $\mu(\boldsymbol{a}_0) = \{\binom{0}{0}\}$, and $[\![\boldsymbol{a}_0]\!] = \Sigma$. The table is not closed, and for this, the learner declares $\boldsymbol{a}_0$ a state, introduces $\boldsymbol{\Sigma}_{\boldsymbol{a}_0} = \{\boldsymbol{a}_1\}$ with $\mu(\boldsymbol{a}_1) = \{\binom{0}{0}\}$, $[\![\boldsymbol{a}_1]\!] = \Sigma$, and $\boldsymbol{R} = \{\boldsymbol{a}_0 \cdot \boldsymbol{a}_1\}$. The resulting table $\boldsymbol{T}_0$ is now closed; the first hypothesis $\boldsymbol{\mathcal{A}}_0$ is constructed.

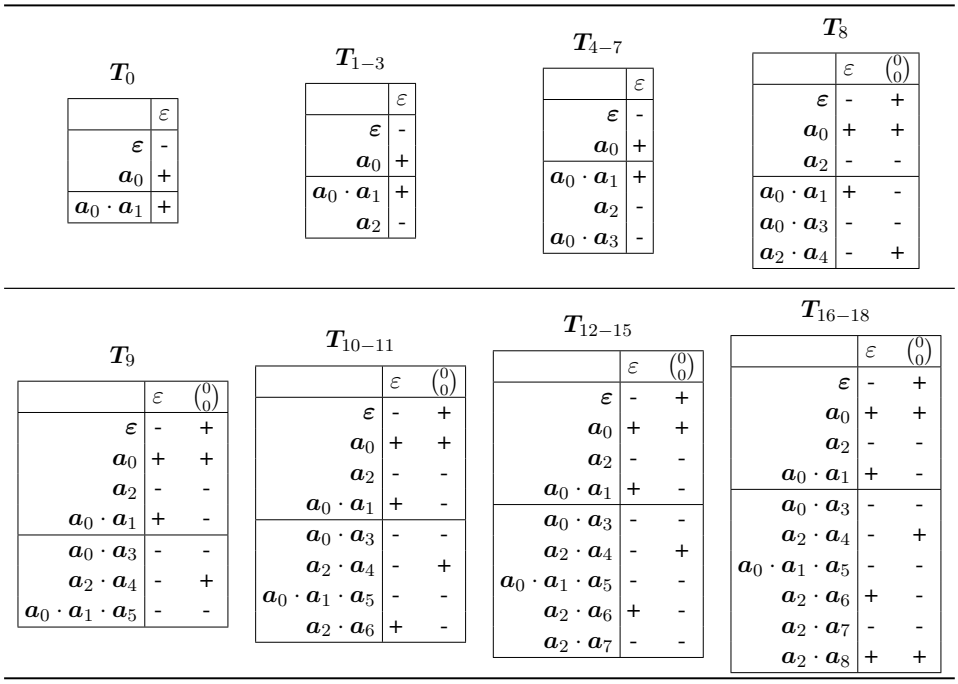


Figure 6.5: Observation tables for Example 6.2.

The counter-example $(\binom{45}{50}, -)$ arrives, to refine the partition at the initial state by introducing a new symbol, and thus to add a new transition to the automaton. The symbolic alphabet is extended to $\boldsymbol{\Sigma}_\varepsilon = \{\boldsymbol{a}_0, \boldsymbol{a}_2\}$ with $[\![\boldsymbol{a}_2]\!] = \{a \in \Sigma : a \geq \binom{45}{50}\}$, $[\![\boldsymbol{a}_0]\!] = \Sigma - [\![\boldsymbol{a}_2]\!]$, and $\mu(\boldsymbol{a}_2) = \{\binom{45}{50}\}$. The new observation table and hypothesis are $\boldsymbol{T}_1$ and $\boldsymbol{\mathcal{A}}_1$, respectively. Two more counter-examples, namely $(\binom{60}{0}, -)$ and $(\binom{0}{70}, -)$, are provided and they refine the partition at the initial state causing a boundary modification. No new state or symbol is added to the automaton, which leaves the structure of the automaton unchanged, see Figure 6.6. However, the semantics function for the initial state updates to ψ_1, ψ_2 and ψ_3 (Figure 6.7), where all letters greater than $\binom{60}{0}$ and $\binom{0}{70}$ are moved to the Σ-semantics of $\boldsymbol{a}_2$. An equivalence query on hypothesis $\boldsymbol{\mathcal{A}}_3$ will result in the counter-example $(\binom{0}{0}\binom{0}{80}, -)$, which, in turn, adds a new symbol $\boldsymbol{a}_3$ to $\boldsymbol{\Sigma}_{\boldsymbol{a}_0}$ and a new transition in the hypothesis.

The three counter-examples that follow, namely, $(\binom{0}{0}\binom{80}{0}, -)$, $(\binom{0}{0}\binom{40}{15}, -)$, and $(\binom{0}{0}\binom{30}{30}, -)$, refine the Σ-semantics for the symbols in $\boldsymbol{\Sigma}_{\boldsymbol{a}_0}$ as shown in ψ_4 though ψ_7. The next counter-example $(\binom{45}{50}\binom{0}{0}, +)$ discovers a new state. Its prefix $\binom{45}{50}$ already exists in $\mu(\boldsymbol{a}_2)$ and $\boldsymbol{a}_2 \in \boldsymbol{R}$, indicating that the prefix $\boldsymbol{a}_2$ is a distinct state. To distinguish it from the initial state, the learner adds the suffix $\binom{0}{0}$ to E. The resulting table $\boldsymbol{T}_8$ is not closed. To render it closed, the prefix $\boldsymbol{a}_0\boldsymbol{a}_1$ moves to $\boldsymbol{S}$. From the resulting table $\boldsymbol{T}_9$, which is both closed and evidence compatible,

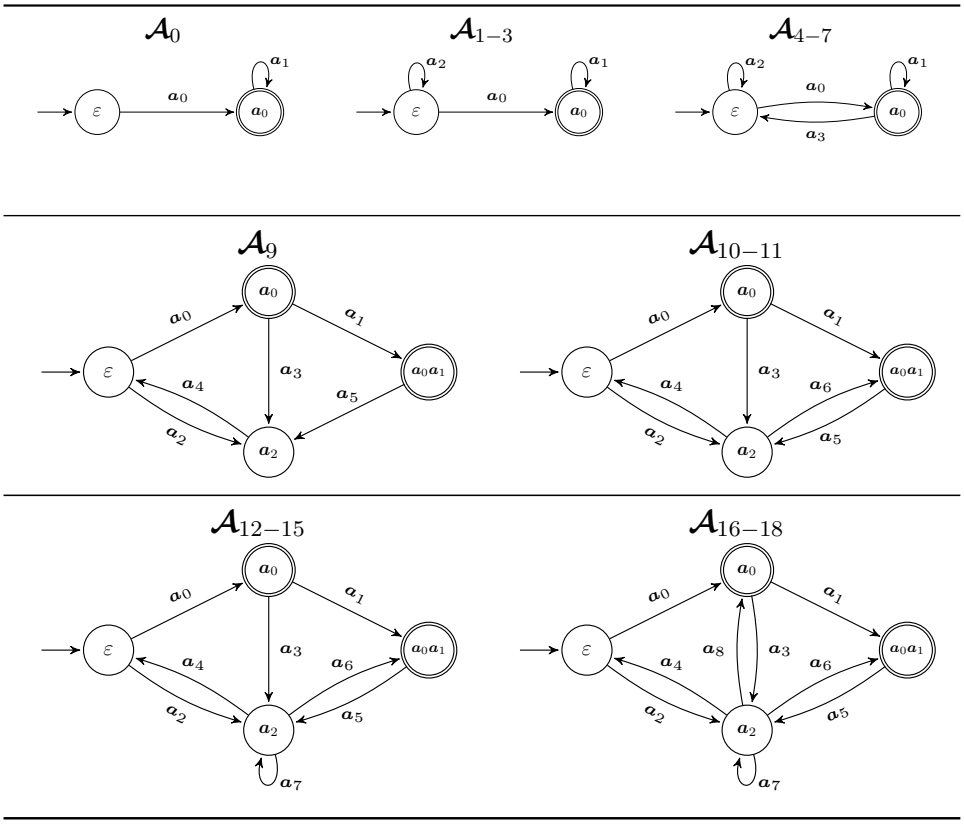


Figure 6.6: Hypothesis automata for Example 6.2.

we conclude the hypothesis $\mathcal{A}_9$. This automaton has four states. The updated Σ-semantics for each state can be seen in ψ_9. Partitions at state a_2 are refined further due to some more counter-examples, see $\mathcal{A}_{10-18}$ and $\psi_{10} - \psi_{18}$ for the updates. The algorithm terminates with $\mathcal{A}_{18}$ as its final hypothesis, after using a total of 20 queries and treating 17 counter-examples. □

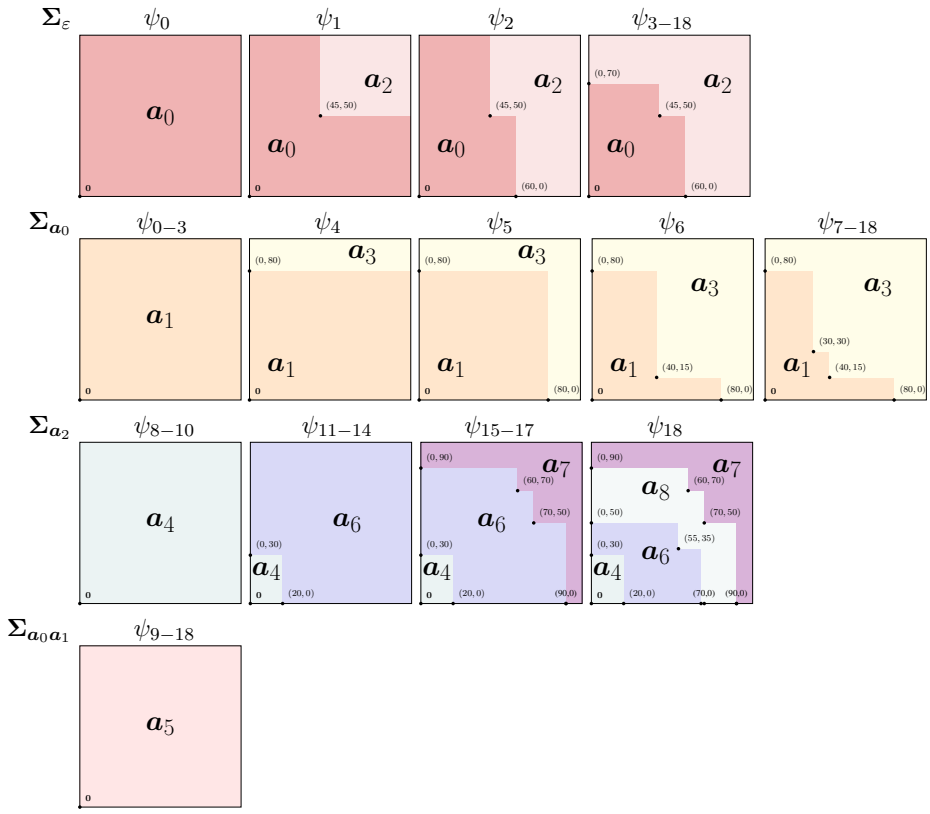


Figure 6.7: Σ-semantics as updated in Example 6.2. The rows correspond to states ε, a_0, a_2 and $a_0 \cdot a_1$, respectively. The function ψ_i refers to the hypothesis $\mathcal{A}_i$.

CHAPTER 7
Learning without a Helpful Teacher

In this chapter, we relax the strong assumption of a helpful teacher. In the new relaxed setting, equivalence queries are approximated by *testing queries*: a call to EQ yields membership queries for a set of randomly selected words; when all of them agree with the hypothesis, the algorithm terminates with a non-zero probability of misclassification; otherwise, we have a counter-example to process. The number of such queries may depend on what we assume about the distribution over Σ^* and what we want to prove about the algorithm, for example, PAC learnability, which is further discussed in Section 8.

The main learning algorithm used in this chapter is Algorithm 8 as it appears in Section 5.3 and counter-examples are treated by applying the symbolic breakpoint method (Procedure 13). In the following, we discuss how we specialize the treatment of evidence incompatibility for alphabets having a particular structure. Specifically, we show alternative methods and convenient modifications that we can apply on the procedures $sample(\Sigma, k)$, $select$ and EVCOMP.

To improve the learning procedure and make it more efficient, we use multiple evidences per state. However, we let only one evidence, the representative, take part in the construction of the observation table and the hypothesis. By doing this we avoid the exponential growth of the concrete sample. The sampling and selection criteria we use to define the evidences and representatives vary and rely upon the type of the alphabet and the partitions. We discuss different types of alphabets and semantics such as interval subsets of countable or uncountable sets and sets of Boolean vectors.

In the previous chapter, the symbolic learning algorithm manages to exactly learn an automaton representation for the target language. Nevertheless, the key to learn such languages, without error, is the presence of a teacher able to answer to equivalence queries and return counter-examples that are minimal. Minimality is what allows the learner to precisely determine the boundaries of each partition.

In this chapter, we assume a teacher who is not that helpful and is able to answer only to membership queries. As we shall see, the symbolic algorithm is able to learn a symbolic automaton to represent the target language, but this time as an approximation. Since minimality of the counter-examples cannot be guaranteed, we choose to use a PAC learning criterion for termination. The set of symbolic recognizable languages is shown to be PAC learnable resulting, almost always, in an automaton that accepts the target language approximately well.

7.1 Approximating the Equivalence Query

We approximate equivalence queries by testing as follows. First, a word $w \in \Sigma^*$ is generated according to some probability distribution $\mathcal{P}$ over Σ^* (line 4). Then a membership query $\text{MQ}(w)$ is posed and the learner checks whether the outcome matches the conjectured hypothesis. If not, the testing procedure terminates returning the word w as a counter-example. If no counter-example has been found after a sufficiently large number of tested words, the equivalence query terminates and the hypothesis is considered correct with some probability of error. The probability distribution that is used to generate words need not be known to the learner.

Let ε and δ be the accuracy and confidence parameters. A hypothesis automaton $\mathcal{A}$ is an ε-approximation of the target L with a probability higher than $1 - \delta$. In such a case we say that the hypothesis is *probably approximately correct* (PAC). For more details on the PAC learnability see Chapter 8.3.2. To guarantee a PAC hypothesis, the testing method should successfully test at least r_i random words, where $r_i = \frac{1}{\varepsilon}\left(\log\frac{1}{\delta} + (i+1)\log 2\right)$, and i is the number of hypotheses that have already been tested.

Procedure 16 Testing Oracle

```
1:  procedure TEST(A^i, ε, δ)                    ▷ ε accuracy, δ confidence
2:      counter = 1
3:      while counter < r_i do
4:          w = random_word_P(Σ*)                ▷ P is a prob. distribution over Σ*
5:          if MQ(w) ≠ A^i(w) then
6:              return w                         ▷ counter-example found
7:          end if
8:          counter = counter + 1
9:      end while
10:     return True
11: end procedure
```

7.2 Learning Languages over $\mathbb{N}, \mathbb{R}$

As in Section 6.1, we consider bounded subsets of $\mathbb{N}$ or $\mathbb{R}$ as the input alphabet Σ, such that the semantics of each symbol is an interval. We disallow disconnected partition blocks, for example, two subsets of even and odd numbers, respectively. Thus, if two disconnected intervals take the same transition, two symbolic letters will be considered. In this setting, the endpoints of an interval associated with a symbolic letter are such that all evidence points between them have the same residual function, while the nearest points outside the interval have different residuals. In an interval setting, semantics can be interpreted as conjunctions of inequalities where the partitioning points reside between two consecutive evidences with different residual functions.

To infer the partition for a state $\boldsymbol{s}$ and define the outgoing transitions in the conjectured automaton we use evidences. Having no prior knowledge of the behavior of a new symbol, when initializing it, we use multiple evidences to get a better approximation. For this reason, when $\boldsymbol{s}$ is a new state, the partition is formed from a sample set of evidences. However, a new evidence may arrive later, introduced by some counter-example, in order to fix an incorrectly conjectured partition.

The initial set of evidences can be any set of k concrete letters denoted by $sample(\Sigma, k)$, see line 3 of Procedure 10. This set can be selected randomly or be the result of a fixed step quantization of Σ, that is, $sample(\Sigma, k) = \{a + i \cdot \Delta : i = 0, \dots\ k-1\}$ where $a = \min \Sigma$ and $\Delta = |\Sigma|/(k-1)$. Moreover, it can be the same each time INITSYMBOL is evoked. One can think of evidence selection by a more adaptive process that depends on the outcome of membership queries. Each symbol admits one element from the set of evidences, usually randomly chosen, as a representative.

Evidence incompatibility appears either right after symbol initialization or after a counter-example treatment. Evidence incompatibility is solved by consecutive calls to EVCOMP until the total incompatibility degree of the observation table becomes zero. Note that partition blocks form convex intervals and, as a consequence, several symbols in the symbolic alphabet may have the same behavior, that is, label a transition that leads to the same target state in the automaton.

For a state $\boldsymbol{s}$, an incompatibility instance at $a \in [\![\boldsymbol{a}]\!]$ indicates either that the partition boundary is imprecise or that we missed a symbol and its corresponding transition. In the first case, the incompatible evidence a appears next to the boundary of the interval and its classification matches the classification of a neighboring symbol $\boldsymbol{a}' \in \mathcal{V}(\boldsymbol{a})$. In this situation, modifying the boundary so that a is moved to $[\![\boldsymbol{a}']\!]$ resolves the incompatibility. On the other hand, when the evidence a is in the interior of an interval, or does not behave as a neighboring symbol in $\mathcal{V}(\boldsymbol{a})$, the incompatibility is resolved by adding a new symbol and refining the existing

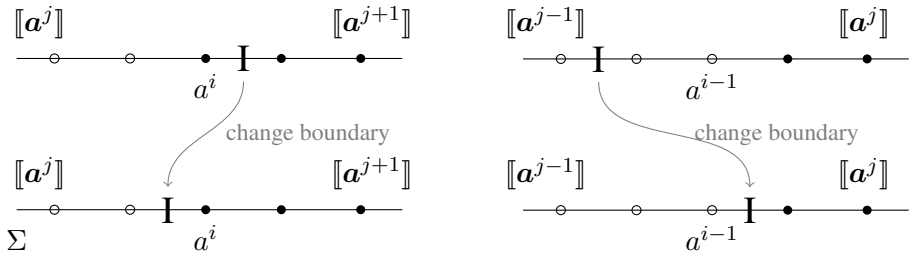


Figure 7.1: Evidence incompatibility solved by boundary modification. The incompatible evidence a^i matches the classification of its neighboring symbol, and a boundary update solves the incompatibility without adding a new symbol.

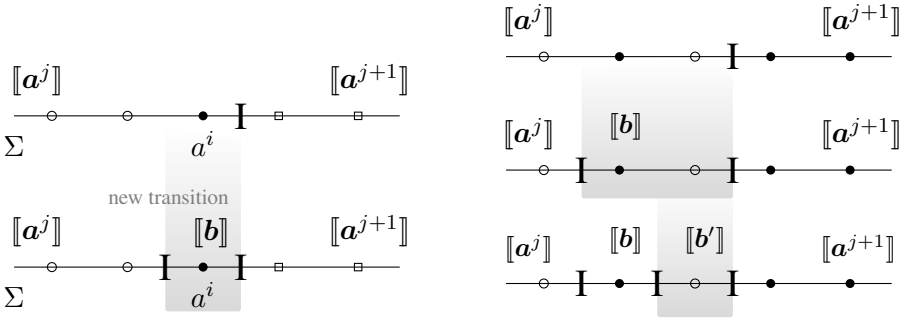


Figure 7.2: Evidence incompatibility solved by introducing new symbols. More symbols may need to be added calling EvComp multiple times until the incompatibility is eliminated.

partition. These two cases are illustrated in Figures 7.1 and 7.2, respectively.

Formally, let $\boldsymbol{s} \in \boldsymbol{S}$ be a state with positive incompatibility degree $\mathrm{M}(\boldsymbol{s}) > 0$, and let $\mu_{\boldsymbol{s}} = \{a^1, \ldots, a^k\} \subset \boldsymbol{S}$ be the set of evidences, ordered such that $a^{i-1} < a^i$ for all i. To simplify notation, f^i denotes the residual $f_{\hat{\mu}(\boldsymbol{s})\cdot a^i}$ when state $\boldsymbol{s}$ is understood from the context.

Let $\boldsymbol{a}^j$ and $\boldsymbol{a}^{j+1}$ denote symbols in $\boldsymbol{\Sigma_s}$ with adjacent semantics, and let $a^{i-1}, a^i \in \mu_{\boldsymbol{s}}$ be two evidences from the same partition block that behave differently, $f^{i-1} \neq f^i$. Moreover, let $\boldsymbol{a}^j \in \boldsymbol{\Sigma_s}$ be the symbol such that $a^{i-1}, a^i \in \mu(\boldsymbol{a}^j)$ where $[\![\boldsymbol{a}^j]\!] = [c, c')$. There exists at least one partitioning point $p \in (a^{i-1}, a^i)$, and it is given by a split method $p = \mathit{split}(a^{i-1}, a^i)$. We let split return the middle point, that is $\mathit{split}(a, a') = (a + a')/2$; other more sophisticated methods can be applied instead, for instance, a split method which asks membership queries in a binary search fashion.

The remedy to evidence incompatibility is to separate a^{i-1} and a^i and map them to different symbols. Procedure 17 describes how evidence incompatibility is solved and how evidences and semantics are updated.

The way this separation is realized, with or without introducing a new symbol,

Procedure 17 Make Evidence Compatible (without Helpful Teacher) - $\mathbb{R}$

procedure EVCOMP
  Let $\boldsymbol{s} \in \boldsymbol{S}$, for which $\mathrm{M}(\boldsymbol{s}) > 0$, where
  $\mu_{\boldsymbol{s}} = \{a^1, \ldots, a^k\}$ such that $a^{i-1} < a^i, \forall i = 2, \ldots, k$
  Let $\boldsymbol{a}^j \in \boldsymbol{\Sigma_s}$, $[\![\boldsymbol{a}^j]\!] = [c, c')$, such that $\exists i : f^{i-1} \neq f^i$ for $a^{i-1}, a^i \in \mu(\boldsymbol{a}^j)$
  $p = split(a^{i-1}, a^i)$ ▷ new partitioning point
  if $f^i = f^{i+1} = \cdots = f^{i+l+1}$, where $a^i, \ldots, a^{i+l} \in \mu(\boldsymbol{a}^j), a^{i+l+1} \in \mu(\boldsymbol{a}^{j+1})$ **then**
    $[\![\boldsymbol{a}^j]\!] = [c, p)$; $[\![\boldsymbol{a}^{j+1}]\!] = [p, c') \cup [\![\boldsymbol{a}^{j+1}]\!]$ ▷ change right frontier
    $\mu(\boldsymbol{a}^{j+1}) = (\mu(\boldsymbol{a}^{j+1}) \cup \mu(\boldsymbol{a}^j)) \cap [\![\boldsymbol{a}^{j+1}]\!]$
    $\mu(\boldsymbol{a}^j) = \mu(\boldsymbol{a}^j) \cap [\![\boldsymbol{a}^j]\!]$
  else if $f^{i-1} = \cdots = f^{i-l}$, where $a^{i-1}, \ldots, a^{i-l+1} \in \mu(\boldsymbol{a}^j), a^{i-l} \in \mu(\boldsymbol{a}^{j-1})$ **then**
    $[\![\boldsymbol{a}^{j-1}]\!] = [\![\boldsymbol{a}^{j-1}]\!] \cup [c, p)$; $[\![\boldsymbol{a}^j]\!] = [p, c')$ ▷ change left frontier
    $\mu(\boldsymbol{a}^{j-1}) = (\mu(\boldsymbol{a}^{j-1}) \cup \mu(\boldsymbol{a}^j)) \cap [\![\boldsymbol{a}^{j-1}]\!]$
    $\mu(\boldsymbol{a}^j) = \mu(\boldsymbol{a}^j) \cap [\![\boldsymbol{a}^j]\!]$
  else
    $\boldsymbol{\Sigma_s} = \boldsymbol{\Sigma_s} \cup \{\boldsymbol{b}\}$ ▷ introduce a new symbol
    $\boldsymbol{R} = \boldsymbol{R} \cup \{\boldsymbol{s} \cdot \boldsymbol{b}\}$
    if $\hat{\mu}(\boldsymbol{a}^j) \leq p$ **then**
      $[\![\boldsymbol{a}^j]\!] = [c, p)$; $[\![\boldsymbol{b}]\!] = [p, c')$
    else
      $[\![\boldsymbol{b}]\!] = [c, p)$; $[\![\boldsymbol{a}^j]\!] = [p, c')$
    end if
    $\mu(\boldsymbol{b}) = \mu(\boldsymbol{a}^j) \cap [\![\boldsymbol{b}]\!]$; $\mu(\boldsymbol{a}^j) = \mu(\boldsymbol{a}^j) \cap [\![\boldsymbol{a}^j]\!]$
    $\hat{\mu}(\boldsymbol{b}) = select(\mu(\boldsymbol{b}))$
    $\boldsymbol{f}_{\boldsymbol{s}\cdot\boldsymbol{b}} = f_{\hat{\mu}(\boldsymbol{s}\cdot\boldsymbol{b})}$
  end if
end procedure

depends on the positions of a^{i-1} and a^i in the set of evidences and the residual values of their neighboring partition blocks. In the following, $\boldsymbol{a}^{j-1}$, $\boldsymbol{a}^j$ and $\boldsymbol{a}^{j+1}$ denote symbols in $\boldsymbol{\Sigma_s}$ with adjacent semantics.

1. *Boundary modification*. Suppose the incompatibility instance is at $a^i \in \mu(\boldsymbol{a}^j)$, and that all other evidences $\mu(\boldsymbol{a}^j)$ to the right of a^i behave like $\min \mu(\boldsymbol{a}^{j+1})$. By changing the partition boundaries and moving a^i from $[\![\boldsymbol{a}^j]\!]$ to $[\![\boldsymbol{a}^{j+1}]\!]$, the incompatibility instance at a^i is eliminated. The new boundary between these two intervals is set to p, see Figure 7.1. The semantics and evidence functions are updated accordingly. The symmetric case, where the incompatibility occurs at $a^{i-1} \in \boldsymbol{a}^j$ with all other evidences of $\mu(\boldsymbol{a}^j)$ on its left behaving like $\max \mu(\boldsymbol{a}^{j-1})$, is treated similarly.

2. *Symbol introduction*. When the above condition does not hold and boundary modification cannot be applied, which happens either when the evidences are internal to $[\![\boldsymbol{a}^j]\!]$, or when their residuals do not match the residuals of any neighboring symbol, the incompatibility is solved by refining the partition. The semantics $[\![\boldsymbol{a}^j]\!]$ is split into two intervals $[c, p)$ and $[p, c')$. A new symbol $\boldsymbol{b}$ is introduced and the interval not containing $\hat{\mu}(\boldsymbol{a}^j)$ is moved from $[\![\boldsymbol{a}^j]\!]$ to $[\![\boldsymbol{b}]\!]$ along with the evidences it contains, see Figure 7.2.

Counter-examples are treated by calling Procedure 13 (see Chapter 5) where each counter-example expands the hypothesis automaton either vertically or horizontally. Let us demonstrate the working of the algorithm in learning a target language over a subset of $\mathbb{R}$.

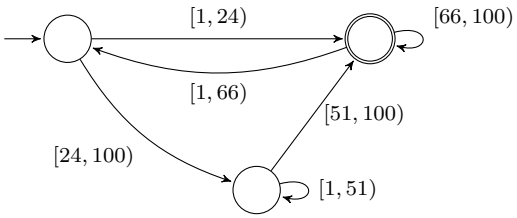


Figure 7.3: Minimal automaton accepting the target language used in Example 7.1.

Example 7.1 (Learning Interval Automata without Minimal Counter-Examples)**.** Let $\Sigma = [0, 100) \subseteq \mathbb{R}$ be the concrete alphabet and a target language $L \subseteq \Sigma^*$, shown in Figure 7.3. We recall that equivalence is approximated by testing and that the teacher answers only membership queries. The observation tables, semantics functions and hypotheses used in this example are shown in Figures 7.4 and 7.5.

The table is initialized with $\boldsymbol{S} = \{\boldsymbol{\varepsilon}\}$ and $E = \{\varepsilon\}$. To determine the alphabet partition at the initial state $\boldsymbol{\varepsilon}$, the learner asks membership queries for the randomly selected one-letter words $\{13, 42, 68, 78, 92\}$. All words in this set except 13 are rejected. Consequently, there are at least two distinct intervals and we take $split(13, 42) = 27$ as their boundary. Each interval is represented by a symbolic letter resulting in $\boldsymbol{\Sigma}_\varepsilon = \{\boldsymbol{a}_1, \boldsymbol{a}_2\}$, $\mu(\boldsymbol{a}_1) = \{13\}$, $\hat{\mu}(\boldsymbol{a}_1) = 13$, $\mu(\boldsymbol{a}_2) = \{42, 68, 78, 92\}$, and $\hat{\mu}(\boldsymbol{a}_2) = 68$. The representatives are randomly chosen from the set of evidences. The semantics, ψ maps all letters smaller than 27 to $\boldsymbol{a}_1$, and maps the rest to $\boldsymbol{a}_2$, that is, $[\![\boldsymbol{a}_1]\!] = [0, 27)$ and $[\![\boldsymbol{a}_2]\!] = [27, 100)$. The table boundary updates to $\boldsymbol{R} = \{\boldsymbol{a}_1, \boldsymbol{a}_2\}$ and the observation table is $\boldsymbol{T}_0$, shown in Figure 7.4.

Table $\boldsymbol{T}_0$ is not closed and in order to fix this, the learner moves $\boldsymbol{a}_1$ to the set of states $\boldsymbol{S}$. To find the possible partitions of Σ at this new state $\boldsymbol{a}_1$, the learner randomly chooses a sample $\{2, 18, 26, 46, 54\}$ of letters and asks membership queries

$\boldsymbol{T}_0$

	ε
$\boldsymbol{\varepsilon}$	$-$
$\boldsymbol{a}_1$	$+$
$\boldsymbol{a}_2$	$-$

$\boldsymbol{T}_1$

	ε
$\boldsymbol{\varepsilon}$	$-$
$\boldsymbol{a}_1$	$+$
$\boldsymbol{a}_2$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_3$	$-$

$\boldsymbol{T}_2$

	ε	11
$\boldsymbol{\varepsilon}$	$-$	$+$
$\boldsymbol{a}_1$	$+$	$-$
$\boldsymbol{a}_2$	$-$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_3$	$-$	$+$

$\boldsymbol{T}_3$

	ε	11
$\boldsymbol{\varepsilon}$	$-$	$+$
$\boldsymbol{a}_1$	$+$	$-$
$\boldsymbol{a}_2$	$-$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_3$	$-$	$+$
$\boldsymbol{a}_2\boldsymbol{a}_4$	$-$	$-$
$\boldsymbol{a}_2\boldsymbol{a}_5$	$+$	$-$

$\boldsymbol{T}_{4-5}$

	ε	11
$\boldsymbol{\varepsilon}$	$-$	$+$
$\boldsymbol{a}_1$	$+$	$-$
$\boldsymbol{a}_2$	$-$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_3$	$-$	$+$
$\boldsymbol{a}_1\boldsymbol{a}_6$	$+$	$-$
$\boldsymbol{a}_2\boldsymbol{a}_4$	$-$	$-$
$\boldsymbol{a}_2\boldsymbol{a}_5$	$+$	$-$

Figure 7.4: Observation tables used in Example 7.1.

concerning the words in $\{13 \cdot 2, 13 \cdot 18, 13 \cdot 26, 13 \cdot 46, 13 \cdot 54\}$. Note that the prefix used here is the representative of $\boldsymbol{a}_1$. The teacher classifies all words as rejected. The new table is $\boldsymbol{T}_1$ with $\boldsymbol{\Sigma}_{\boldsymbol{a}_1} = \{\boldsymbol{a}_3\}$, $\mu(\boldsymbol{a}_3) = \{2, 18, 26, 46, 54\}$, $\hat{\mu}(\boldsymbol{a}_3) = 18$, and $[\![\boldsymbol{a}_3]\!] = [0, 100)$. The new table is closed and the first hypothesis $\boldsymbol{\mathcal{A}}_1$ is conjectured.

The hypothesis is tested on a set of words, randomly chosen from some distribution, typically unknown to the learner. After some successful tests, a word $35 \cdot 52 \cdot 11$ is found, which is accepted by $\boldsymbol{\mathcal{A}}_1$ but is outside the target language. The learner takes this word as a counter-example and analyzes it using the symbolic breakpoint method. At iteration $i = 2$ of Procedure 13, condition (5.3) is violated, in particular $\text{MQ}(\hat{\mu}(\boldsymbol{\varepsilon} \cdot \boldsymbol{a}_2) \cdot 11) = \text{MQ}(68 \cdot 11) \neq flag = +$. Thus, the suffix 11 is added as a distinguishing word to E. The observation table $\boldsymbol{T}_2$ obtained after adding the new suffix is, as expected, not closed. The table is made closed by letting $\boldsymbol{a_2}$ be a new state, resulting in table $\boldsymbol{T}_3$, where $\boldsymbol{\Sigma}_{\boldsymbol{a}_2} = \{\boldsymbol{a}_4, \boldsymbol{a}_5\}$, $\mu(\boldsymbol{a}_4) = \{17, 27\}$, $\hat{\mu}(\boldsymbol{a}_4) = 17$, $[\![\boldsymbol{a}_4]\!] = [0, 45)$, $\mu(\boldsymbol{a}_5) = \{64, 72, 94\}$, $\hat{\mu}(\boldsymbol{a}_5) = 72$ and $[\![\boldsymbol{a}_5]\!] = [45, 100)$. The corresponding new conjecture is $\boldsymbol{\mathcal{A}}_3$.

Automaton $\boldsymbol{\mathcal{A}}_3$ is tested and a counter-example $12 \cdot 73 \cdot 4$ is provided. The breakpoint method discovers that condition (5.2) is violated because letter 73 is not part of the semantics of $\boldsymbol{a}_3$. This letter is added as a new evidence to $\mu(\boldsymbol{a}_3)$. The evidence incompatibility is solved by splitting the existing partition into two subintervals. A new symbol $\boldsymbol{a}_6$ is added to $\boldsymbol{\Sigma}_{\boldsymbol{a}_1}$, such that $\mu(\boldsymbol{a}_6) = \{73\}$ and $[\![\boldsymbol{a}_6]\!] = [63, 100)$. The new observation table and hypothesis automaton are $\boldsymbol{T}_4$ and $\boldsymbol{\mathcal{A}}_4$, respectively.

The next counter-example $52 \cdot 47$, also adds a new evidence, this time to symbol $\boldsymbol{a}_5$. The classification of the new evidence matches the classification of $\boldsymbol{a}_4$, which is a neighboring symbol. The boundary between $[\![\boldsymbol{a}_4]\!]$ and $[\![\boldsymbol{a}_5]\!]$ is moved from 45 to 55, thus resolving the evidence incompatibility. The new hypothesis $\boldsymbol{\mathcal{A}}_5$ is suc-

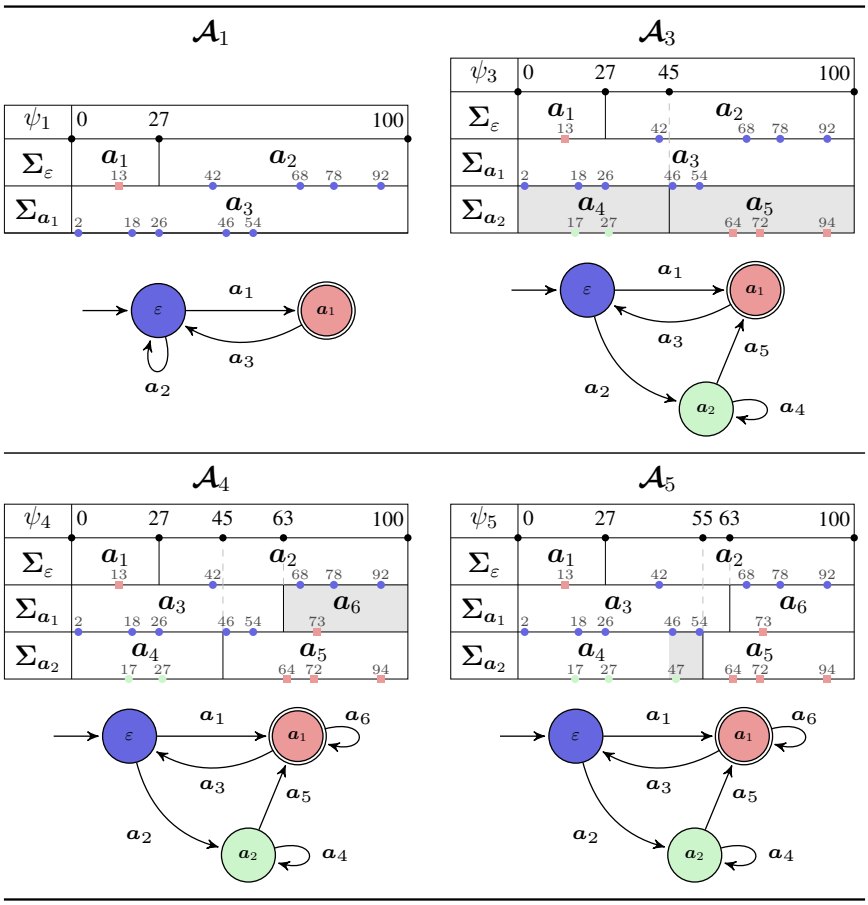


Figure 7.5: Symbolic automata and semantics function learned in Example 7.1.

cessfully tested without discovering any other counter-example and the algorithm terminates while returning $\mathcal{A}_5$ as an answer. □

7.3 Learning Languages over $\mathbb{B}^n$

We demonstrate the versatility of the algorithm, which was first developed for numerical alphabets, by adapting it to languages over the alphabet $\Sigma = \mathbb{B}^n$ of Boolean vectors accessed by variables $\{x_1, \ldots, x_n\}$. All components of the algorithm remain the same except the construction of alphabet partitions and their modification due to evidence incompatibility. These should be adapted to the particular nature of the Boolean hyper-cube. The concrete semantics of the symbolic letters in a state s, which can either form sub-cubes (terms) or finite unions of sub-cubes, will be defined by a function $\psi_s : \mathbb{B}^n \to \mathbf{\Sigma}_s$.

At any given moment, the raw data for inducing the alphabet partition at s is the

Procedure 18 Make Evidence Compatible (without Helpful Teacher) - $\mathbb{B}^n$

procedure EVCOMP
  Let $\boldsymbol{s} \in \boldsymbol{S}$ be a state for which $\mathrm{M}(\boldsymbol{s}) > 0$
  UPDATE($T_{\boldsymbol{s}}$) ▷ make tree consistent with sample
  for all $h \in \mathcal{F}_{\boldsymbol{s}}$ **do** ▷ Update $\xi_{\boldsymbol{s}}$
    if $\exists \boldsymbol{a} \in \boldsymbol{\Sigma}_{\boldsymbol{s}}$ s.t. $h = f_{\hat{\mu}(\boldsymbol{s})\cdot\hat{\mu}(\boldsymbol{a})}$ **then**
      ▷ h is already associated with an existing symbol $\boldsymbol{a}$
      $\xi_{\boldsymbol{s}}(h) = \boldsymbol{a}$
      $\mu(\boldsymbol{a}) = \{a^i \in \mu_{\boldsymbol{s}} : f^i = h\}$
    else ▷ h does not match any pre-existing residual
      $\boldsymbol{\Sigma}_{\boldsymbol{s}} = \boldsymbol{\Sigma}_{\boldsymbol{s}} \cup \{\boldsymbol{b}\}$ ▷ introduce new symbol $\boldsymbol{b}$
      $\boldsymbol{R} = \boldsymbol{R} \cup \{\boldsymbol{s}\cdot\boldsymbol{b}\}$
      $\xi_{\boldsymbol{s}}(h) = \boldsymbol{b}$
      $\mu(\boldsymbol{b}) = \{a^i \in \mu_{\boldsymbol{s}} : f^i = h\}$
      $\hat{\mu}(\boldsymbol{b}) = select(\mu(\boldsymbol{b}))$
    end if
  end for
  $\psi_{\boldsymbol{s}} = \xi_{\boldsymbol{s}} \circ T_{\boldsymbol{s}}$
end procedure

sample $\{(a^i, f^i) : a^i \in \mu_{\boldsymbol{s}}\}$ where $\mu_{\boldsymbol{s}}$ is the set of all evidences for state $\boldsymbol{s}$, and for every a^i, $f^i = f_{\hat{\mu}(\boldsymbol{s})\cdot a^i}$ is the residual associated with a^i. Let $\mathcal{F}_{\boldsymbol{s}} = \{f^i : a^i \in \mu_{\boldsymbol{s}}\}$ be the set of all observed distinct residuals associated with the one-letter successors of $\boldsymbol{s}$. We view $\psi_{\boldsymbol{s}}$, associated with a state $\boldsymbol{s} \in \boldsymbol{S}$, as a composition of two functions, that is, $\psi_{\boldsymbol{s}} = \xi_{\boldsymbol{s}} \circ T_{\psi_{\boldsymbol{s}}}$, where $T_{\psi_{\boldsymbol{s}}} : \mathbb{B}^n \to \mathcal{F}_{\boldsymbol{s}}$ and $\xi_{\boldsymbol{s}} : \mathcal{F}_{\boldsymbol{s}} \to \boldsymbol{\Sigma}_{\boldsymbol{s}}$. To simplify the notation, we use $T_{\boldsymbol{s}}$ instead of $T_{\psi_{\boldsymbol{s}}}$.

The function $T_{\boldsymbol{s}}$ is represented by a binary decision tree (see Section 2.2) whose leaf nodes are labeled by elements of $\mathcal{F}_{\boldsymbol{s}}$ and which is considered compatible with the sample if it agrees with it on the elements of $\mu_{\boldsymbol{s}}$. The function $\xi_{\boldsymbol{s}}$ can be seen as a relabeling that maps each distinct residual to a symbol from $\boldsymbol{\Sigma}_{\boldsymbol{s}}$. We let this function be a bijection. Had we wanted to follow the "convex" partition approach that we used for numerical alphabets, we should have associated a fresh symbol with each leaf node of the tree, thus letting $[\![\boldsymbol{a}]\!]$ be a cube/term for every $\boldsymbol{a} \in \boldsymbol{\Sigma}_q$. However, unlike numerical alphabets, we prefer here to associate the same symbol with multiple leaf nodes that share the same label (residual), allowing the semantics of a symbol to be a finite union of cubes. This results in $|\boldsymbol{\Sigma}_{\boldsymbol{s}}| = |\mathcal{F}_{\boldsymbol{s}}|$ and the existence of at most one symbol that labels a transition between any pair of states. For the function $\xi_{\boldsymbol{s}}$ to be well defined, we require that it sends each representative

to the symbol it represents, i.e.,

$$\xi_{\boldsymbol{s}}(f_{\hat{\mu}(\boldsymbol{s})\cdot\hat{\mu}(\boldsymbol{a})}) = \boldsymbol{a} \text{ for all symbols } \boldsymbol{a} \in \boldsymbol{\Sigma_s}. \tag{7.1}$$

To learn the semantics $\psi_{\boldsymbol{s}}$ for state $\boldsymbol{s}$, it suffices to learn the two functions $\xi_{\boldsymbol{s}}$ and $T_{\boldsymbol{s}}$. We first build $T_{\boldsymbol{s}}$ as a decision tree where all evidences mapped to the same leaf node agree on their residual function. Hence, learning alphabet partitions is an instance of learning decision trees using algorithms such as CART [BFSO84], ID3 [Qui86], or ID5 [Utg89] that construct a tree compatible with a labeled sample. Then $\xi_{\boldsymbol{s}}$ may be updated and new symbols may be added to $\boldsymbol{\Sigma_s}$ if necessary.

Evidence incompatibility in a state $\boldsymbol{s}$ appears when the decision tree $T_{\boldsymbol{s}}$ is not compatible with the sample. This may happen in three occasions during the execution of the algorithm, the first being symbol initialization. Recall that when a new state $\boldsymbol{s}$ is introduced, we create a new symbol $\boldsymbol{a}$ and collect evidences for it, which may have different residuals while being associated with the same single root node. The second occasion occurs when new evidence is added to a symbol, making a leaf node in the tree impure. Finally, when some new suffix is added to E, the set $\mathcal{F}_{\boldsymbol{s}}$ of distinct residuals (rows in the table) may increase and the labels of existing evidences may change.

The simplest way to fix incompatibility of a decision tree is to split impure leaf nodes until purification. However, this may lead to very deep trees and it is preferable to reconstruct the tree each time the sample is updated in a way that leads to incompatibility. In the simple (second) case where a new evidence is added, we can use an incremental algorithm such as ID5, see Section 2.3, which restructures only parts of the tree that need to be modified, leaving the rest of the tree intact. This algorithm produces the same tree as a non-incremental algorithm would, while performing less computation. In the third case, we build the tree from scratch and this is also what we do after initialization where the incremental and non-incremental algorithms coincide.

Once a tree $T_{\boldsymbol{s}}$ is made compatible with the sample, the semantics of the symbolic alphabet, expressed via $\psi_{\boldsymbol{s}}$, is updated. This is nothing but updating $\xi_{\boldsymbol{s}}$ to map the possibly new residuals from $\mathcal{F}_{\boldsymbol{s}}$ to $\Sigma_{\boldsymbol{s}}$. First, with each symbol $\boldsymbol{a}$ that already exists, we re-associate the leaves of $T_{\boldsymbol{s}}$ that agree with the labels of its representative such that (7.1) holds. Note that, $\mu_{\boldsymbol{s}}$ may have changed but the representative of an existing symbol remains the same. Then, in the case where the set $\mathcal{F}_{\boldsymbol{s}}$ of distinct residuals has increased in number, we introduce a new symbolic letter for each new residual and select its representative.

The whole process is described in Procedure 18. We use UPDATE$(T_{\boldsymbol{s}})$ for any tree learning algorithm that can be used to make the tree compatible with the sample. This can be an incremental or non-incremental algorithm as the ones presented

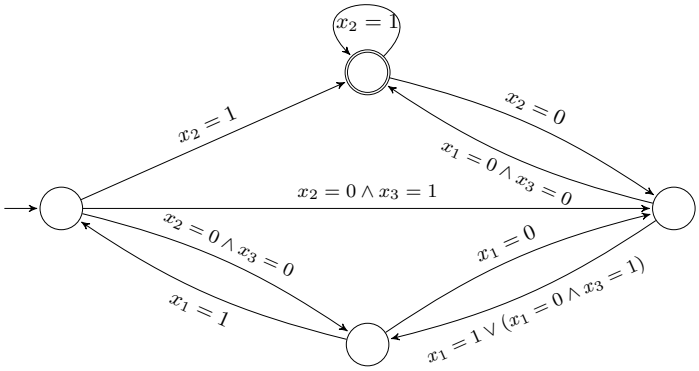


Figure 7.6: Minimal automaton accepting the target language used in Example 7.2.

in Section 2.3 and need not be specified here.

Example 7.2 (Learning Languages over Boolean Alphabets)**.** In this example, the algorithm is applied to learn the target language L over $\Sigma = \mathbb{B}^4$, which is shown in Figure 7.6. We denote by $X = \{x_1, x_2, x_3, x_4\}$ the set of Boolean variables used to access the attributes of the alphabet. All tables encountered during the execution of the algorithm are shown in Figure 7.7, and the semantics functions $\psi_{\boldsymbol{s}}$, realized by decision trees, appear in Figure 7.9 in the form of Karnaugh maps. All letters that are used as evidences in the example are noted in Figure 7.9, where different colors and shapes indicate different residuals.

The learner starts by initializing the observation table to $\boldsymbol{T}_0$. Like every new state, the initial state $\boldsymbol{\varepsilon}$ admits one outgoing transition. Let us label this transition by the symbol $\boldsymbol{a}_0$ and let the symbolic alphabet be $\boldsymbol{\Sigma}_{\boldsymbol{\varepsilon}} = \{\boldsymbol{a}_0\}$. Since $\boldsymbol{a}_0$ represents, initially, all concrete letters in Σ, the semantics function $\psi^0_{\boldsymbol{\varepsilon}}$ is a decision tree consisting of a single node, and $[\![\boldsymbol{a}_0]\!] = \Sigma$. To find the behavior of $\boldsymbol{a}_0$, a set of concrete letters is sampled and used as the evidence for it, e.g., $\mu(\boldsymbol{a}_0) = \{(0000), (0010), (1011), (1000), (1101)\}$. The letter $\hat{\mu}(\boldsymbol{a}_0) = (0000)$ is chosen as a representative. The table is filled in after all evidences are queried.

As not all evidences behave the same, e.g., evidence $(1101) \in \mu(\boldsymbol{a}_0)$ behaves differently than the representative (0000), the observation table $\boldsymbol{T}_0$ is not evidence compatible and thus the partition at state $\boldsymbol{\varepsilon}$ needs refinement. The learner applies the tree induction algorithm CART, which is used throughout this example, to refine the BDT $\psi_{\boldsymbol{\varepsilon}}$, and finds that Σ is best split into two blocks based on the values of variable x_2. A new symbol $\boldsymbol{a}_1$, which is added to $\boldsymbol{\Sigma}_{\boldsymbol{\varepsilon}}$, is used to denote the new block (the new behavior). With this in mind, all letters for which $x_2 = 0$ are mapped to $\boldsymbol{a}_0$ and all letters for which $x_2 = 1$ are mapped to $\boldsymbol{a}_1$. The semantics function for $\boldsymbol{\varepsilon}$ updates to $\psi^1_{\boldsymbol{\varepsilon}}$.

$\boldsymbol{T}_0$

	ε
$\boldsymbol{\varepsilon}$	$-$
$\boldsymbol{a}_0$	$-$

$\boldsymbol{T}_1$

	ε
$\boldsymbol{\varepsilon}$	$-$
$\boldsymbol{a}_0$	$-$
$\boldsymbol{a}_1$	$+$

$\boldsymbol{T}_2$

	ε
$\boldsymbol{\varepsilon}$	$-$
$\boldsymbol{a}_1$	$+$
$\boldsymbol{a}_0$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_2$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_3$	$+$

$\boldsymbol{T}_3$

	ε	0000
$\boldsymbol{\varepsilon}$	$-$	$-$
$\boldsymbol{a}_1$	$+$	$-$
$\boldsymbol{a}_0$	$-$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_2$	$-$	$+$
$\boldsymbol{a}_1\boldsymbol{a}_3$	$+$	$-$

$\boldsymbol{T}_4$

	ε	0000
$\boldsymbol{\varepsilon}$	$-$	$-$
$\boldsymbol{a}_1$	$+$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_2$	$-$	$+$
$\boldsymbol{a}_0$	$-$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_3$	$+$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_4$	$-$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_6$	$+$	$-$

$\boldsymbol{T}_{5-6}$

	ε	0000
$\boldsymbol{\varepsilon}$	$-$	$-$
$\boldsymbol{a}_1$	$+$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_2$	$-$	$+$
$\boldsymbol{a}_0$	$-$	$-$
$\boldsymbol{a}_5$	$-$	$+$
$\boldsymbol{a}_1\boldsymbol{a}_3$	$+$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_4$	$-$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_6$	$+$	$-$

$\boldsymbol{T}_7$

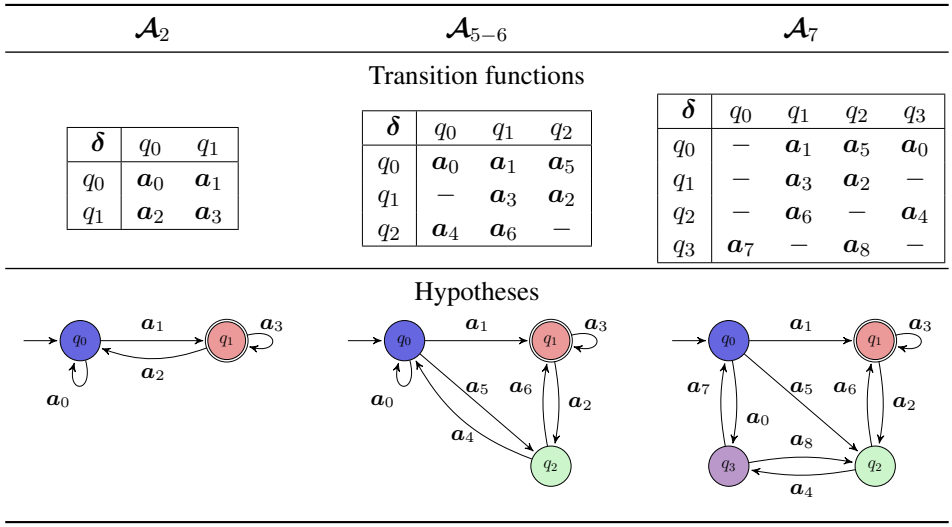
	ε	0000	1110
$\boldsymbol{\varepsilon}$	$-$	$-$	$+$
$\boldsymbol{a}_1$	$+$	$-$	$+$
$\boldsymbol{a}_1\boldsymbol{a}_2$	$-$	$+$	$-$
$\boldsymbol{a}_0$	$-$	$-$	$-$
$\boldsymbol{a}_5$	$-$	$+$	$-$
$\boldsymbol{a}_0\boldsymbol{a}_7$	$-$	$-$	$+$
$\boldsymbol{a}_0\boldsymbol{a}_8$	$-$	$+$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_3$	$+$	$-$	$+$
$\boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_4$	$-$	$-$	$-$
$\boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_6$	$+$	$-$	$+$

Figure 7.7: Observation tables generated during the execution of the algorithm on Example 7.2.

The resulting observation table $\boldsymbol{T}_1$ is not closed. To close it, $\boldsymbol{a}_1$ becomes a state. The symbolic alphabet for $\boldsymbol{a}_1$ is set initially to $\boldsymbol{\Sigma}_{\boldsymbol{a}_1} = \{\boldsymbol{a}_2\}$. The evidence for the new state/symbol is sampled to $\mu(\boldsymbol{a}_1) = \{(0000), (0010), (1011), (1000), (1101)\}$ and $\hat{\mu}(\boldsymbol{a}_1) = (0000)$ is chosen as a representative. For simplicity, we use the same sample for each new state, but this is not a requirement in applying the algorithm. After solving evidence incompatibility that is caused by the symbol initialization, we obtain the symbolic alphabet $\boldsymbol{\Sigma}_{\boldsymbol{a}_1} = \{\boldsymbol{a}_2, \boldsymbol{a}_3\}$, the semantics function $\psi = \{\psi^1_{\boldsymbol{\varepsilon}}, \psi_{\boldsymbol{a}_1}\}$, and the observation table $\boldsymbol{T}_2$. The first conjectured automaton is $\boldsymbol{\mathcal{A}}_2$, shown in Figure 7.8.

After testing the hypothesis $\boldsymbol{\mathcal{A}}_2$, a counter-example $w = (1010) \cdot (0000)$ has been found. The learner, applying the breakpoint method, detects the distinguishing word (0000) and adds it to E. After posing the necessary MQ's, the new observation table is $\boldsymbol{T}_3$. The table is neither closed nor evidence compatible. We first close $\boldsymbol{T}_3$ and then solve the incompatibility at $\boldsymbol{\varepsilon}$ which was caused by changing the residual functions of the evidence, see $\psi^2_{\boldsymbol{\varepsilon}}$.

The table is made closed by making $\boldsymbol{a}_1\boldsymbol{a}_2$ a state. After sampling evidence,



δ	q_0	q_1
q_0	a_0	a_1
q_1	a_2	a_3

δ	q_0	q_1	q_2
q_0	a_0	a_1	a_5
q_1	$-$	a_3	a_2
q_2	a_4	a_6	$-$

δ	q_0	q_1	q_2	q_3
q_0	$-$	a_1	a_5	a_0
q_1	$-$	a_3	a_2	$-$
q_2	$-$	a_6	$-$	a_4
q_3	a_7	$-$	a_8	$-$

Figure 7.8: Intermediate and final Automata conjectures made in Example 7.2.

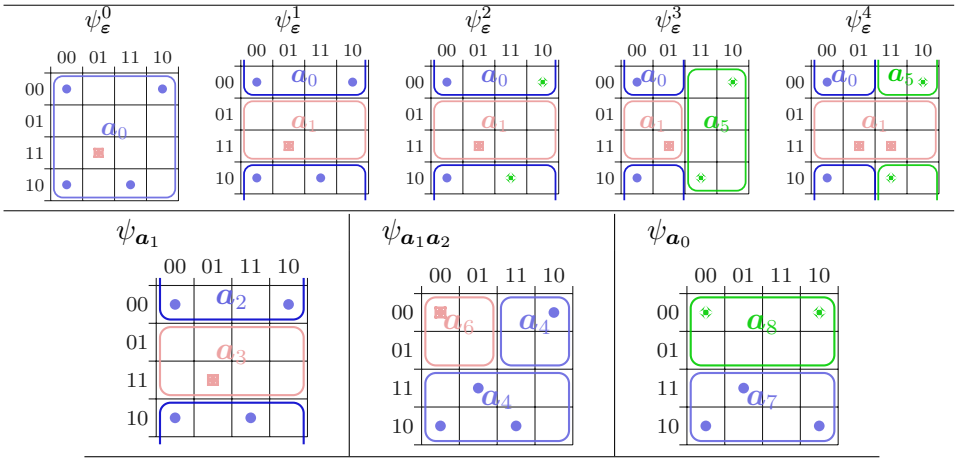


Figure 7.9: Semantics functions used in Example 7.2. We show the evolution of ψ_ε over time, while for the other states we show only the final partition. We use symbols such as $\{\bullet, \blacksquare, \blacklozenge\}$ to indicate different residuals.

selecting the representative, and building an evidence compatible partition at $\boldsymbol{a}_1\boldsymbol{a}_2$, defined by $\psi_{\boldsymbol{a}_1\boldsymbol{a}_2}$, the table updates to $\boldsymbol{T}_4$. Now, we return to state $\boldsymbol{\varepsilon}$ and solve the evidence incompatibility that appeared in $\psi^2_{\boldsymbol{\varepsilon}}$ when (0000) was added to E. The decision tree $\psi_{\boldsymbol{\varepsilon}}$ is reconstructed from scratch to become compatible with the updated sample. The symbols are rearranged such that they match the residuals of their representatives. A new symbol $\boldsymbol{a}_5$ is added to the symbolic alphabet. The new evidence compatible partition is $\psi^3_{\boldsymbol{\varepsilon}}$.

The new table is $\boldsymbol{T}_5$. The new hypothesis $\boldsymbol{\mathcal{A}}_5$ is tested for equivalence, providing the counter-example $w = (1111)$. Applying the breakpoint method on the new counter-example shows that the letter (1111) does not behave like $\boldsymbol{a}_5$ as it was assumed in the hypothesis. The letter (1111) is, thus, added to $\mu(\boldsymbol{a}_5)$ as a new evidence, causing once more an incompatibility at the initial state. The BDT $\psi_{\boldsymbol{\varepsilon}}$ is fixed and updated to $\psi^4_{\boldsymbol{\varepsilon}}$. Since this incompatibility is due to a new evidence and not to an update of the whole sample, the tree is updated using an incremental algorithm. Observe that this last counter-example fixes the partition by rearranging the sub-cubes that form the partition of $\mathbb{B}^4$ without adding any new transition. The observation table remains unchanged.

Testing the next hypothesis $\boldsymbol{\mathcal{A}}_6$, provides the counter-example $w = (1000) \cdot (1000)\cdot(0000)\cdot(0000)\cdot(1110)$. The treatment of the counter-example adds the new suffix (1110) to E. This distinguishes the initial state from the prefix $\boldsymbol{a}_0$, which is now identified as a state. The partition at state $\boldsymbol{a}_0$, after refining and making evidence compatible, is given by $\psi_{\boldsymbol{a}_0}$, and the observation table $\boldsymbol{T}_7$ is obtained. The new final hypothesis $\boldsymbol{\mathcal{A}}_7$ is tested without discovering any counter-example and the algorithm terminates.

CHAPTER 8

Theoretical Analysis: Complexity and Termination

In this chapter, we study the termination and complexity of the algorithm, first under the assumption of a helpful teacher and then in the random testing setting. Most of the analysis resembles that of L^* except for the special feature of symbolic learning, the modification of partition boundaries without adding new states and transitions, which requires a special treatment. We also propose one way to define probabilities on $\mathbf{\Sigma}^*$ and show how to compute the probability of a misclassification error based on the symmetric difference between the target language and a hypothesis.

8.1 Updating the Hypothesis: Counter-Examples

Let $L \subseteq \Sigma^*$ be the target language and let $\boldsymbol{\mathcal{A}}$ be the minimal symbolic automaton recognizing L such that $L = L(\boldsymbol{\mathcal{A}})$. We assume that $\boldsymbol{\mathcal{A}}$ has n states and at most m outgoing transitions per state, that is, a symbolic alphabet $\mathbf{\Sigma} = \biguplus_q \mathbf{\Sigma}_q$ such that $|\mathbf{\Sigma}_q| \leq m$ for every state q.

A typical run of the algorithm produces a sequence $\{\boldsymbol{\mathcal{A}}_i\}$ of hypotheses with each $\boldsymbol{\mathcal{A}}_i$ different from $\boldsymbol{\mathcal{A}}_{i-1}$, see Figure 8.1. We say that $\boldsymbol{\mathcal{A}}_i$ is *expansive* if it is different from $\boldsymbol{\mathcal{A}}_{i-1}$ in the transition graph, that is, $\boldsymbol{\mathcal{A}}_i$ has more states and/or transitions than $\boldsymbol{\mathcal{A}}_{i-1}$. When the structure of the transition graph does not change and only the partition boundaries differ, we call $\boldsymbol{\mathcal{A}}_i$ a *non-expansive* hypothesis. Our goal is to study the behavior of this sequence under different assumptions and in particular to show under what conditions it is finite. Moreover, we show under which conditions exact learnability can be replaced by PAC learnability.

Finite number of expansions. Our learning algorithm produces, by construction,

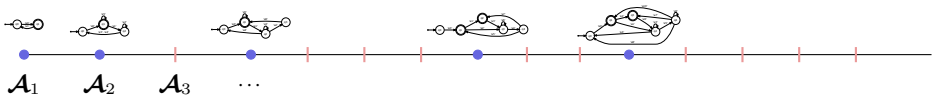


Figure 8.1: A sequence of hypothesis automata $\{\mathcal{A}_i\}$ produced during a run of the algorithm. We use the symbols $\{\bullet\ ,\ |\ \}$ to denote expansive and non-expansive hypotheses, respectively.

minimal automata and the size of the hypothesis cannot exceed the size of the symbolic automaton $\mathcal{A}$. Consequently, the number of expansive hypotheses is bounded by $\mathcal{O}(mn)$, because at most $n-1$ involve the discovery of a new state and at most $n(m-1)$ involve a new transition. What remains to be shown is that the number of non-expansive hypotheses is bounded. This type of hypothesis is highly influenced by the type of teacher and his ability to answer equivalence queries correctly, as well as the nature of the alphabet. Before moving on to study the special cases of teachers and alphabets, we explain in more detail how a boundary modification affects the hypothesis. Then, it is important to define the error of a hypothesis and how to compute this.

Boundary modifications. To infer the partitions, the learner uses partial information and hence, boundaries are often only approximated. As a result, a word may be misclassified just by the fact that it involves a letter falling on the wrong side of the boundary. Such a word is then provided as a counter-example. The misclassified letter is added and yields evidence incompatibility, which is solved by modifying the existing boundary. This procedure may not affect the structure of the hypothesis automaton.

Let us consider a partition at some state of the hypothesis. Figure 8.2 shows such an example with a one dimensional numerical alphabet. Let the real boundary p between two neighboring intervals be approximated by $p' = split(a, b)$, where a and b are evidences with different residuals. At this point the $split(a, b)$ method, which always results a point in $[a, b]$, is not specified. The letters in $[p, p']$ are wrongly classified as equivalent to a. We call this interval the *boundary error*. This boundary is modified when a new evidence b' is added to $[p, p']$. The partitioning point p' updates to $p'' = split(a, b')$ in the next hypothesis.

8.2 Hypothesis Error

Let $L \subseteq \Sigma^*$ be the target language and let $\mathcal{A}$ be a conjectured automaton accepting the language $L_{\mathcal{A}}$. The language $L_{\mathcal{A}}$ is not equivalent to L when some states or transitions are still missing, or there exists a partition that contains some boundary error.

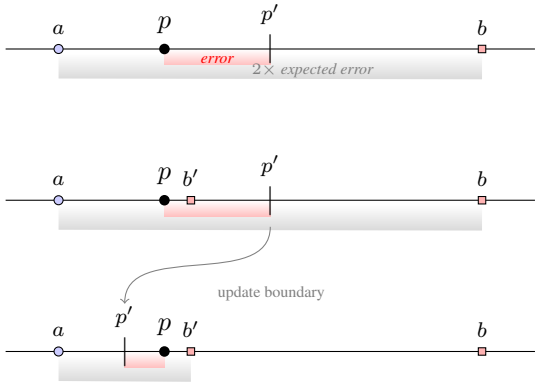


Figure 8.2: A partition boundary where p, the real partition boundary, is approximated by $p' = split(a, b)$, where a and b are evidences with different residuals. This is the case where $p < p'$ and for the case $p' < p$ we get some a' in the counter-example.

The quality of $\mathcal{A}$ is determined by its distance from L, denoted by $d(L, L_{\mathcal{A}})$. One way[1] to define this distance is to measure the probability of error, that is, the probability of choosing a misclassified word $w \in \Sigma^*$ that belongs to one language but not to the other. This definition includes all possible sources of error, i.e., incomplete structure, boundary errors.

The set of all misclassified words is exactly the *symmetric difference* of the languages L and $L_{\mathcal{A}}$, i.e., $L \oplus L_{\mathcal{A}} = L \setminus L_{\mathcal{A}} \uplus L_{\mathcal{A}} \setminus L$. Hence, we define the distance as

$$d(L, L_{\mathcal{A}}) = \mathcal{P}(L \oplus L_{\mathcal{A}}),$$

where $\mathcal{P}$ is a probability distribution over Σ^*.

When an automaton representation of the target language is available, one can find the probability of the error by first computing the symmetric difference of the target and conjectured languages, and then computing the probability of this language. To compute the probability we use the *relative volumes* that are described below.

8.2.1 A Probability Distribution on Σ^*

In this section we define a natural probability distribution $\mathcal{P}$ over the set of all words Σ^*, where Σ is an input alphabet. First we choose some distribution $\mathcal{P}_\ell : \mathbb{N} \to [0, 1]$, which provides the probability of a word w to be of certain length. Then, each letter in the word obeys a uniform distribution over the alphabet, that is $\mathcal{P}_\Sigma(a) = 1/|\Sigma|$ for all $a \in \Sigma$. The probability of a given word $w = a_1 \cdots a_k \in \Sigma^*$

[1] Other ways to measure the error (or distance of two languages) have been proposed in the literature, see for instance [SVVV14].

can then be defined as

$$\mathcal{P}(a_1 \cdots a_k) = \mathcal{P}_\ell(k) \prod_{i=1}^{k} \mathcal{P}_\Sigma(a_i) = \frac{\mathcal{P}_\ell(k)}{|\Sigma|^k}.$$

As usual, probability distributions $\mathcal{P}_\Sigma$ and $\mathcal{P}$ extend to sets of letters and words, respectively. Thus,

$$\mathcal{P}_\Sigma(I) = \sum_{a \in I} \mathcal{P}_\Sigma(a) = |I|/|\Sigma|, \text{ and}$$

$$\mathcal{P}(L) = \sum_{w \in L} \mathcal{P}(w),$$

where $I \subseteq \Sigma$ and $L \subseteq \Sigma^*$.

By partitioning the set of words according to their length, the probability of a language can be written as

$$\begin{aligned} \mathcal{P}(L) &= \sum_{k=0}^{\infty} \mathcal{P}\big(L \cap \Sigma^k\big) \\ &= \sum_{k=0}^{\infty} \mathcal{P}_\ell(k) \cdot \mathcal{P}\big(L \mid \Sigma^k\big), \end{aligned} \tag{8.1}$$

where $\mathcal{P}\big(L \big| \Sigma^k\big)$ is the probability that a word belongs to L given its length is k. We call the last probability the *relative k-volume* of L and denote it as $D_k(L)$. In the following we show how to compute these relative k-volumes using stochastic matrices.

8.2.2 Computing the Relative Volumes

The *relative k-volume* of a language $L \subseteq \Sigma^*$, where $k \geq 0$ is an integer, is defined as $D_k(L) = \mathcal{P}(L|\ \Sigma^k) = vol(L_k)/vol(\Sigma^k)$ where $vol(\Sigma^k) = |\Sigma|^k$ and $vol(L_k)$ is the volume of $L_k = L \cap \Sigma^k$, which is the subset of L that contains only the words of length k. The volume $vol(L_k)$ is defined and computed recursively by value iteration on the structure of the automaton that recognizes L [ABD15].

Let $\boldsymbol{\mathcal{A}} = (\Sigma, \boldsymbol{\Sigma}, \psi, Q, \boldsymbol{\delta}, q_0, F)$ be a symbolic automaton such that $L = L(\boldsymbol{\mathcal{A}})$. We can define on this automaton, a *volume transition matrix* $V = (\nu_{ij}) \in M_{|Q|}(\mathbb{R})$, where ν_{ij} denotes the volume of direct transitions from state i to state j in $\boldsymbol{\mathcal{A}}$ defined as $\nu_{ij} = |\{a \in \Sigma : \boldsymbol{\delta}(i, \psi(a)) = j\}|$. In the symbolic automata setting, this is equivalent to $\nu_{ij} = |\cup_{\boldsymbol{a} \in \boldsymbol{\Sigma}_i, \boldsymbol{\delta}(i,\boldsymbol{a})=j} [\![\boldsymbol{a}]\!]|$. Intuitively, this indicates the number of possible ways there are for going from state i to state j in one step. Note that, for

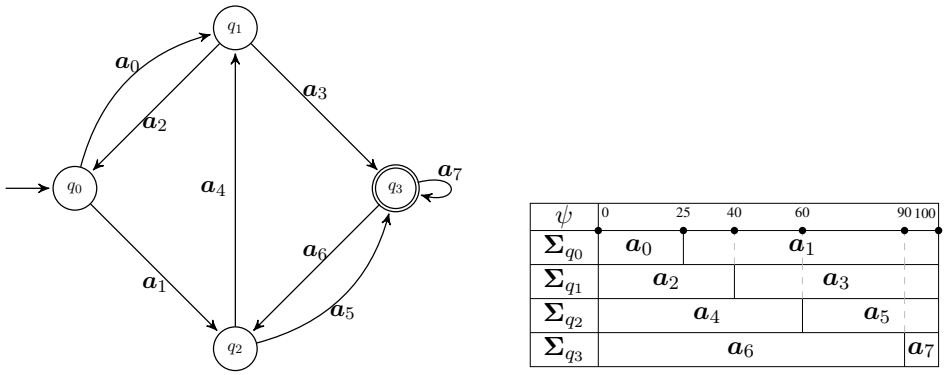


Figure 8.3: A symbolic automaton $\boldsymbol{\mathcal{A}}$ and its semantics.

a complete and deterministic automaton $\boldsymbol{\mathcal{A}}$, $\sum_{j=0}^{|Q|} \nu_{ij} = vol(\Sigma) = |\Sigma|$ for every $i \in Q$, and hence $V/|\Sigma|$ forms a right stochastic matrix. We can find all powers of the matrix V, such that V^0 is the identity matrix, $V^1 = V$, and $V^n = V^{n-1}V$ for all $n > 1$.

The element $\nu_{ij}^{(n)}$, which denotes the (i, j) entry of the matrix V^n, gives the number of possible paths of length n that lead from a state i to j. The volume of L_k is given by

$$vol(L_k) = \sum_{j \in F} \nu_{q_0 j}^{(k)},$$

where q_0 is the initial state and F is the set of accepting states in $\boldsymbol{\mathcal{A}}$.

We illustrate this computation with an example. Let $\Sigma = [0, 100) \subseteq \mathbb{R}$ be an alphabet and let $L \subseteq \Sigma^*$ be a language, accepted by the symbolic automaton $\boldsymbol{\mathcal{A}}$ shown in Figure 8.3. Each state q_i in the automaton has two outgoing transitions labeled by $\boldsymbol{a}_{2i}$ and $\boldsymbol{a}_{2i+1}$, respectively. Given that the volume of an interval $[a, b)$ is $vol([a, b)) = b - a$, the volume transition matrix defined by $\boldsymbol{\mathcal{A}}$ is

$$V = \begin{pmatrix} 0 & vol(\boldsymbol{a}_0) & vol(\boldsymbol{a}_1) & 0 \\ vol(\boldsymbol{a}_2) & 0 & 0 & vol(\boldsymbol{a}_3) \\ 0 & vol(\boldsymbol{a}_4) & 0 & vol(\boldsymbol{a}_5) \\ 0 & 0 & vol(\boldsymbol{a}_6) & vol(\boldsymbol{a}_7) \end{pmatrix} = \begin{pmatrix} 0 & 25 & 75 & 0 \\ 40 & 0 & 0 & 60 \\ 0 & 60 & 0 & 40 \\ 0 & 0 & 90 & 10 \end{pmatrix}.$$

To find the volume of L_k we compute the matrix V^k and sum up all numbers of the first row of matrix V^k that correspond to final states. In the example this is the last element of the first row. In Figure 8.4, we present some powers of the volume transition matrix, where the values are already normalized. According to this figure, we see that $vol(L_1) = 0$ and $vol(L_2) = 45$, etc., and that $D_2(L) = 0.45$, $D_{45}(L) = 0.315$.

$$V^0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad V^3/|\Sigma^3| = \begin{pmatrix} 0.18 & 0.025 & 0.48 & 0.315 \\ 0.04 & 0.5040 & 0.054 & 0.4020 \\ 0 & 0.276 & 0.54 & 0.184 \\ 0.216 & 0.054 & 0.333 & 0.3970 \end{pmatrix}$$

$$V^2/|\Sigma^2| = \begin{pmatrix} 0.1 & 0.45 & 0 & 0.45 \\ 0 & 0.1 & 0.84 & 0.06 \\ 0.24 & 0 & 0.36 & 0.4 \\ 0 & 0.54 & 0.09 & 0.37 \end{pmatrix} \qquad V^n/|\Sigma^n| = \begin{pmatrix} 0.0945 & 0.2362 & 0.3543 & 0.3150 \\ 0.0945 & 0.2362 & 0.3543 & 0.3150 \\ 0.0945 & 0.2362 & 0.3543 & 0.3150 \\ 0.0945 & 0.2362 & 0.3543 & 0.3150 \end{pmatrix}$$
for all $n > 30$.

Figure 8.4: Some powers of the volume transition matrix V defined by $\mathcal{A}$ of Figure 8.3.

8.3 Complexity and Termination

The termination and complexity of the algorithm are highly affected by the nature of the teacher, that is, the way equivalence queries are realized. We considered two types of equivalence checking. The first is a helpful teacher (Chapter 6), which will always provide a counter-example when there exists one, and moreover, this counter-example is minimal. The second (Chapter 7) implements equivalence by random sampling from the same distribution used to compute the error. This implementation may, of course, miss some counter-examples, but some of its runs may also have the opposite effect and produce a series of counter-examples that reduce the error as little as possible. In the case where $\boldsymbol{\Sigma}$ is finite this may lead to a behavior as bad as that of a concrete algorithm and in the case of alphabet $\mathbb{R}$, even to non termination. However, as we will argue, under probabilistic assumptions, the probability of such a scenario is zero.

8.3.1 Using a Helpful Teacher (Minimal Counter-Examples)

First, we examine the case where a helpful teacher is present. Such a teacher, as presented in Chapter 6, returns a minimal counter-example whenever a conjecture is wrong. The complexity of the symbolic algorithm is influenced not by the size of the alphabet but by the resolution (partition size) with which we observe it.

Concerning the size of the observation table $\boldsymbol{T}$, the set of prefixes $\boldsymbol{S}$ monotonically increases until it reaches the size of exactly n elements. Since the table, by construction, is always kept reduced, the elements in $\boldsymbol{S}$ represent exactly the states of the automaton. The size of the boundary cannot exceed the total number of transitions in the automaton, and thus it is bounded by the size $mn - n + 1$. The number of suffixes in E, playing a distinguishing role for the states of the automaton, range between $\log_2 n$ and n. Taking all these into account, we can conclude that the size of the table ranges between $(n + m) \log_2 n$ and $n(mn + 1)$.

Regarding the size of the samples, and in consequence the number of membership queries, we discuss separately totally and partially ordered alphabets.

In the case of numerical totally-ordered alphabets, the learner uses Procedure 14 to treat counter-examples. Partitions are defined as closed-open intervals, the number of sufficient evidence, which is needed to define a partition block, is exactly one, the minimal letter belonging to the interval. The evidence incompatibility, caused by a new evidence, is immediately solved by splitting the current partition block and introducing a transition. As only one evidence is used per transition, the size of the concrete sample coincides with the size of the symbolic sample associated with the table, and the number of membership queries needed is $\mathcal{O}(mn^2)$.

Moreover, due to minimality of the counter-examples in the length-lexicographic order, choosing as evidence the minimal element of a partition block the boundaries are correctly defined and allow no error. This way, the run of the algorithm does not contain any non-expansive hypotheses. The algorithm terminates after all states and transitions have been discovered and defined. Hence, for termination at most $\mathcal{O}(mn)$ equivalence queries are used in total, from which at most $n-1$ of them add a new state to the hypothesis, and at most $n(m-1)$ add a transition.

When the alphabet is partially-ordered, we have seen in Section 6.2 that each partition block F_i may need multiple evidences in order to be fully defined. For this, some additional queries are asked. We know that when we use monotone partitions, as in Section 6.2, a finite (at most l) number of incomparable minimal points is sufficient to fully define F_i. Hence, for every row in $\boldsymbol{T}$, at most $(l-1)$ supplementary words are added to the concrete sample. The number of membership queries is thus $\mathcal{O}(n^2ml)$.

Furthermore, in the case of partially-ordered alphabets, additional counter-examples may be given in order to refine existing partitions. That is, at most $l-1$ additional evidences are returned per partition block. This implies that the number of non-expansive hypotheses is limited to $nm(l-1)$. We conclude that the number of equivalence queries is bounded by $\mathcal{O}(mn^2)$ if $l < n$ and $\mathcal{O}(lmn)$ otherwise.

Proposition 8.1. *Let $L \subseteq \Sigma^*$ be a target language. The symbolic learning algorithm, which uses a helpful teacher, terminates with a correct conjecture after asking at most $\mathcal{O}(n^2ml)$ membership queries and at most $\mathcal{O}(lmn)$ equivalence queries.*

8.3.2 Equivalence using Random Tests

The algorithm implements equivalence checks by comparing membership in L and in $L_{\mathcal{A}}$ for words randomly selected according to a probability distribution $\mathcal{P}$ over Σ^*. The procedure that performs equivalence testing is shown in Procedure 16. It takes as input a hypothesis $\mathcal{A}_i$ and the accuracy and confidence parameters and

returns either a True statement or a counter-example. Note that testing does not guarantee the discovery of a counter-example when the conjecture is wrong.

Within this setting, the learner can only guarantee to return an approximation of the target language as there is no way to ensure exact learnability. We use the PAC (probably approximately correct) learnability criterion instead, where the learner terminates with an automaton, which is almost always a good approximation of the target language L. The approximation measure that is used is the distance between the conjecture language and the target, see Section 8.2.

Definition 8.2 (PAC Learning [Val84])**.** A learning algorithm learns a language L in a probably-approximately correct (PAC) manner with probability parameters ϵ (accuracy) and δ (confidence) if its output $\boldsymbol{\mathcal{A}}$ satisfies

$$Pr(\mathcal{P}(L \oplus L_{\boldsymbol{\mathcal{A}}}) \leq \epsilon) \geq 1 - \delta, \tag{8.2}$$

The minimum number of tests sufficient to conclude that a hypothesis $\boldsymbol{\mathcal{A}}$ satisfies (8.2) depends on the accuracy and confidence parameters as well as the number of previous hypotheses that are made. This number was introduced in [Ang87, Ang88] for the class of automata and applies in a straightforward way to the symbolic case.

Proposition 8.3. *The symbolic learning algorithm PAC-learns a language L if the i-th equivalence query tests $r_i = \frac{1}{\epsilon}(\ln \frac{1}{\delta} + (i+1)\ln 2)$ random words without finding a counter-example.*

Proof. Let $\boldsymbol{\mathcal{A}}$ be the i-th hypothesis tested. We show that after testing $r_i = \frac{1}{\epsilon}(\ln \frac{1}{\delta} + (i+1)\ln 2)$ random words, then (8.2) holds, or equivalently,

$$Pr(\mathcal{P}(L \oplus L_{\boldsymbol{\mathcal{A}}}) \geq \epsilon) \leq \delta.$$

Taking into account that a) the probability of choosing a word which does not belong to the symmetric difference is $(1-p)$ when the probability of finding a counter-example is p, and b) all words are independently drawn from the same probability distribution, then not drawing a counter-example after r_i draws implies that

$$Pr(\mathcal{P}(L \oplus L_{\boldsymbol{\mathcal{A}}}) \geq \epsilon) \leq (1-\epsilon)^{r_i} \leq \sum_{i=1}^{\infty}(1-\epsilon)^{r_i} \leq \sum_{i=1}^{\infty} e^{-\epsilon r_i} \leq \sum_{i=1}^{\infty} \frac{\delta}{2^{i+1}} \leq \delta$$

□

As for termination, it is always guaranteed when the alphabet is finite, but when Σ is a subset of $\mathbb{R}$ there is a theoretical possibility of an infinite sequence of non-

expansive hypotheses. We give below an informal argument for the improbability of such a behavior of the algorithm.

In concrete automata, a sample whose prefixes contains all words of size up to $2n$ will exercise all the paths in the automaton and is sufficient to discover all states and transitions. In the symbolic setting, the same holds for valid symbolic words and since all the transition guards are intervals of a positive measure, we will obtain such a sample after a finite number of equivalence queries. Hence we can restrict our discussion to a sequence of non-expansive hypotheses applied to the final automaton.

Secondly, we have the following about what happens when we modify an error interval $[p, p']$ associated with a boundary between two neighboring intervals.

Proposition 8.4. *Each time an error interval $[p, p']$ is modified, the upper bound on its size is divided by at least 2*

Proof. Let $a, b \in \Sigma$ be two consecutive evidences with different residuals in the neighboring intervals. Let $p \in [a, b]$ be the real boundary which is approximated by $p' = split(a, b)$, as shown in Figure 8.2. Without loss of generality, let $p < p'$. The boundary error satisfies $|p - p'| \leq \frac{|a-b|}{2}$. Now, let $c \in [p, p']$ be an evidence indicating that p' is wrong. It updates the approximation to $p'' = split(a, c)$. The upper bound on the error is reduced at least to half, $|p - p''| \leq \frac{|a-c|}{2} \leq \frac{|a-b|}{4}$. □

Naturally, this will decrease by two the bound on the probability to get a counter-example involving this interval in a subsequent step. To have an infinite run of the algorithm requires that each equivalence query (which involves r_i tests) yields a counter-example. Each such counter-example modifies at least one error interval and halves the bounds on its size. It can be shown, based on probability estimates, that the sizes of *all* boundary intervals decrease with some rate which is at least quadratic in the number of tests, and so does the probability of finding a counter-example. This rate of decrease is faster then the linear growth in the number of queries due to the r_i formula and, hence, the scenario of an infinite sequence of equivalence queries has a probability zero.

9 Empirical Results

The symbolic learning algorithm has been implemented and applied to several case studies. In this chapter, we first give some insights on the implementation of the algorithms and procedures in Section 9.1. Then in Section 9.2, we present a simple example of learning a language over a subset of the reals, which illustrates the behavior of the algorithm, especially in the setting of PAC learnability. In Section 9.3, we compare the algorithm's performance to the three concrete learning algorithms presented in Chapter 3. For the comparison to be feasible, it is necessary to restrict ourselves to languages whose alphabet is finite. Another case study appears in Section 9.4, where the target languages represent valid types of passwords that are defined over the ASCII characters. Finally, we study the symbolic algorithm for Boolean vectors and compare its performance with the numerical case in Sections 9.5 and 9.6, respectively.

9.1 General Comments on the Implementation

All algorithms presented in this thesis have been implemented using Python, in a prototype implementation. Preserving the notation used in the previous chapters, the method $sample(\Sigma, k)$ in Procedure 10 returns a sample of size $k = 5$ chosen uniformly from Σ, and the method $select(A)$ chooses one representative from a uniform distribution over a set of evidences A. The $split(a, b)$ method, used in Procedure 17, returns the middle point of the interval (a, b), where $a, b \in \mathbb{N}$ or $\mathbb{R}$.

The teacher is implemented separately and it can be used by all types of learners. A teacher is characterized as *helpful* of not, depending on the way equivalence is checked. In both cases, we let H denote the hypothesis automaton given to the teacher. For a helpful teacher, the target language $L \subseteq \Sigma^*$ is given as a DFA or a symbolic automaton $\mathcal{A}$. Then, the teacher computes the symmetric difference

$L(A) \oplus L(H)$ and checks emptiness. It returns a *true* statement if this language is empty. Otherwise, it finds the shortest accepting paths in the automaton and returns the lexicographically minimal word in these as a counter-example.

On the other hand, a non-helpful teacher checks equivalence through testing. In this case, the target language need not necessarily be represented as an automaton. To check equivalence, the membership of random words is compared for H and L. The sufficient number of tested words is set to $\frac{1}{\epsilon}(\ln \frac{1}{\delta} + (i+1)\ln 2)$ where ϵ and δ are parameters given by the user, and i is a counter for the number of hypotheses made so far.

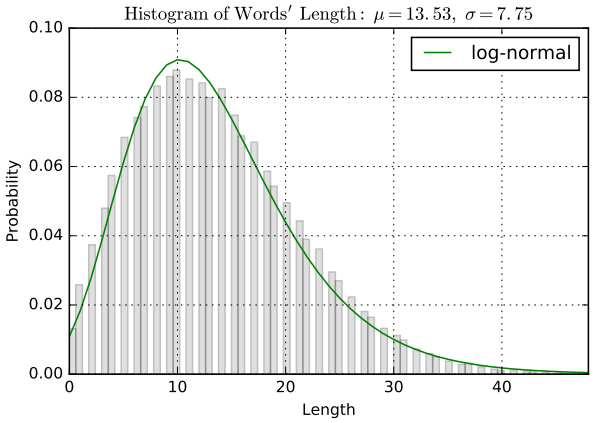


Figure 9.1: The log-normal distribution $\Lambda(13.53, 7.75^2)$ along with its normalization over the natural numbers. This distribution is used for the experiments presented in Section 9.4.

To perform testing we implemented an oracle that produces random words from Σ^*. Since determining a probability distribution on Σ^* is not a trivial task, we implemented this by combining two distributions (see Section 8.2.1). The first distribution $\mathcal{P}_\ell$, defined over $\mathbb{N}$, determines the length of the random word; while the second distribution $\mathcal{P}_\Sigma$, defined over the concrete alphabet Σ, selects the letters that fill each position in the word. In order to prevent very long counter-examples, it is preferable to use a right skewed distribution to determine the length of the random words. We use the *log-normal distribution*[1] in our experiments, normalized over the natural numbers; Figure 9.1 shows the log-normal distribution, which is used for the experiments in Section 9.4. In all our experiments, we use a uniform distribution over the alphabet Σ.

It is important to state here that all comparisons that appear in the sequel pose

[1] The *log-normal distribution* is given by $\Lambda(x; \mu, \sigma^2) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{\ln x - \mu}{\sigma}\right)^2}$, where the variable $x > 0$ and the parameters μ and $\sigma > 0$ are real numbers. Intuitively, one can think that if u is distributed according to the normal distribution $N(\mu, \sigma^2)$ and $u = \ln x$, then x is distributed according to the log-normal distribution $\Lambda(\mu, \sigma^2)$.

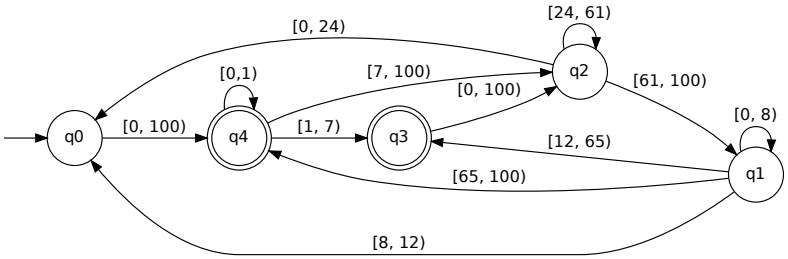


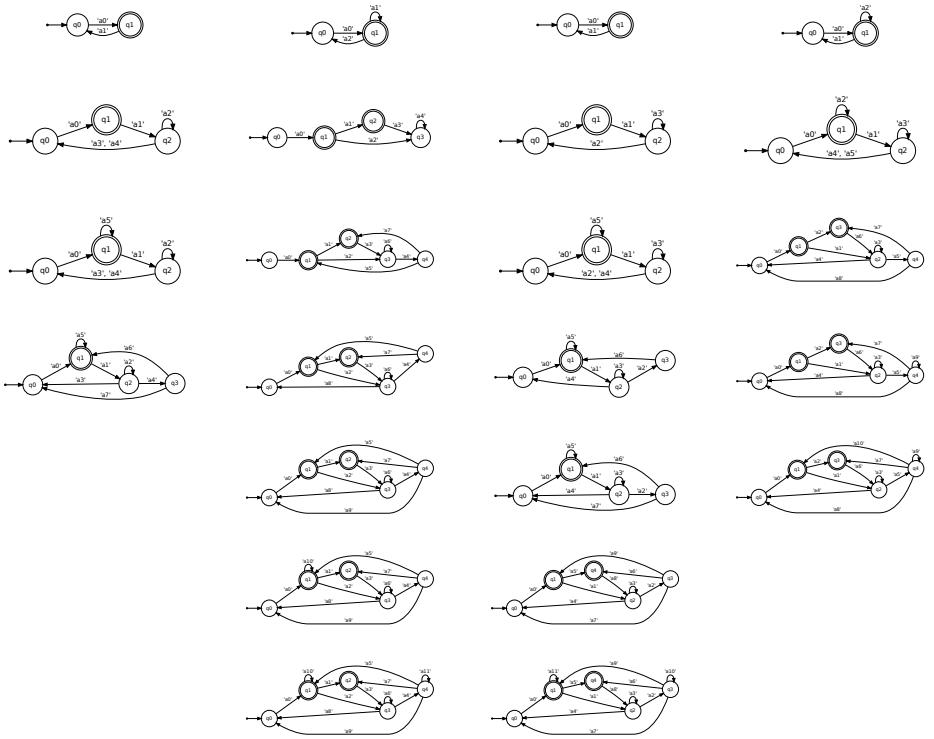
Figure 9.2: The minimal automaton accepting the target language $L \subseteq \Sigma^*$, where $\Sigma = [0, 100) \subseteq \mathbb{R}$, used for the empirical evaluation in Section 9.2.

queries to the same teacher and they are compared on learning the same target languages. Any distribution over Σ^* that is used during the testing phase does not change until a learner terminates, and the error is measured using this same distribution. The performance is measured in terms of time, number of queries and size of the final observation table and the samples.

9.2 On the Behavior of the Symbolic Learning Algorithm

When the equivalence query is realized by random testing, different runs of the same algorithm may produce slightly different hypotheses. In this section, our aim is to show how different counter-examples impact the evolution of the learning process. We observe the final hypothesis as well as the intermediate conjectures made by the learner and discuss the performance of the symbolic algorithm according to the automaton size, the number of queries and the approximation error. We apply the symbolic algorithm to the target language $L \subseteq \Sigma^*$, accepted by the automaton in Figure 9.2, where the concrete input alphabet is $\Sigma = [0, 100) \subset \mathbb{R}$. We set the accuracy and confidence parameters to $\epsilon = \delta = 0.05$. For our discussion, we have selected four representative runs that appear in Figure 9.3.

The symbolic algorithm manages to recover the whole structure of the target automaton in the majority of the runs, allowing a small error in the partition boundaries, see for instance, the runs $R2$ and $R3$. However, we also observe runs, such as $R4$, where the final error is small, less than 0.2% as required, even though there are missing transitions or states in the final hypothesis. Finally, since PAC learnability is used as the termination criterion, the algorithm terminates with probability 95% ($\delta = 5\%$) with a hypothesis with a total error greater than $\epsilon = 5\%$. This happens in run $R1$, where the learner failed to discover all states of the target, resulting in an error 0.0528. Figure 9.4 shows how the error evolves for all hypotheses made along the learning path for each run.



	$\lvert\boldsymbol{S}\rvert$	$\lvert\boldsymbol{\Sigma}\rvert$	$\lvert\boldsymbol{M_T}\rvert$	$\lvert M_{\boldsymbol{T}}\rvert$	MQ's	EQ's	tests	error (D)
Run $R1$	4	8	36	83	153	10	750	5.28%
Run $R2$	5	12	39	116	1052	13	2040	0.17%
Run $R3$	5	12	52	149	820	13	1346	0.26%
Run $R4$	5	11	36	105	1653	10	4235	0.17%

Figure 9.3: The evolution of the conjectured automata in four different runs of the symbolic algorithm. The target language is accepted by the minimal automaton shown in Figure 9.2. Columns $|\boldsymbol{S}|$ and $|\boldsymbol{\Sigma}|$ denote the number of states and transitions learned. The size of the symbolic and concrete samples appear in columns $|\boldsymbol{M_T}|$ and $|M_{\boldsymbol{T}}|$, respectively; MQ's denotes the total number of membership queries used to fill the table, to fix evidence incompatibility and to treat counter-examples. The number of equivalence queries, as well as the total number of words tested, are shown in columns EQ's and *tests*, respectively. The last column shows the approximation error of the final hypothesis.

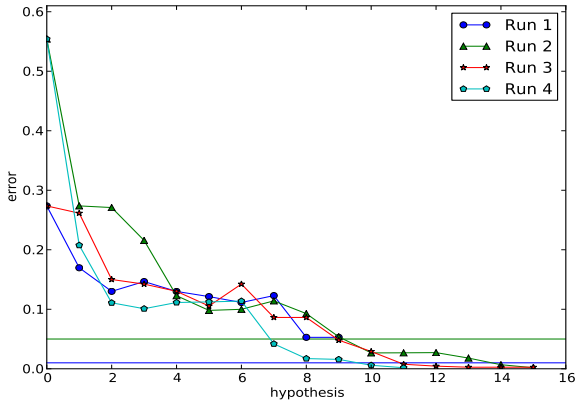


Figure 9.4: The error of each hypothesis made during the runs of the symbolic algorithm described in Section 9.2.

9.3 Comparison with Other Algorithms

In this section, we compare the performance of the symbolic learning algorithm (SL) with the three non-symbolic learning algorithms presented and discussed in Chapter 3. Namely, the L^* (L) algorithm [Ang87], the reduced L^* (LR) [RS93], and the suffixes variation of L^* (LM) taken from [MP95].

The four algorithms are tested on learning the same target languages. The input alphabet Σ, which is adapted to fit the needs of the concrete algorithms, is a finite subset of $\mathbb{N}$. All algorithms have access to the same non-helpful teacher and use the same membership and equivalence queries. The target languages, provided as minimal symbolic automata, are randomly generated. The accuracy and confidence parameters are kept fixed to $\epsilon = \delta = 0.05$.

We compare the performance of the different algorithms based on two main criteria, the size of the input alphabet and the size of target automaton. To this end, we perform two groups of experiments. In the first group, the four learning algorithms are asked to learn target languages that share a fixed structure of a minimal automaton that consists of 15 states and a maximum of 5 outgoing transitions per state. Then the alphabet size ranges from 2 to 200. The partitions of the alphabet at each state are randomly generated. An example of such an automaton appears in Figure 9.5. In the second group, we are interested in testing the effect of the automaton size. For this, we fix the alphabet to $\Sigma = \{0, \ldots, 150\}$ and generate automata of a random structure. We let the number of states, in their minimal representation, to range from 3 to 45 and allow a maximal number of 5 partition blocks per state.

The different algorithms are tested and compared in terms of several size and

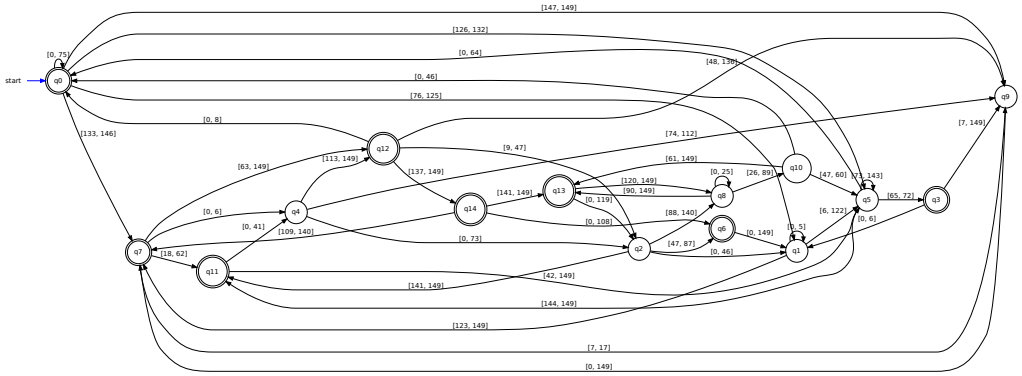


Figure 9.5: An example of an automaton used in the comparison of Section 9.3, where the automaton structure is kept fixed while the alphabet is increasing and the partitions are randomly generated.

time aspects. All results are measured and averaged over multiple runs for each target automaton. Specifically, for each different size of alphabet (in the first group) and each size of automaton (in the second group), we generate 10 different target languages and apply each algorithm 10 times.

We measured the total number of equivalence queries needed until reaching a final hypothesis, the number of intermediate conjectures made by the learner, the number of counter-examples found, as well as the total number of tests performed to guarantee the validity of a hypothesis. Moreover, the number of membership queries is measured, counting separately the queries that are used to characterize the sample, to treat the counter-examples, and used during the testing phase. In addition to the final size of the table (prefixes, boundary, suffixes) and the size of the samples, we also compare the number of states and transitions discovered upon termination. The quality of each final hypothesis is measured by mean of the error as defined in Chapter 8. Finally, we compare the relative timings, such as the total execution time, the total time spent for testing, and the time spent in treating counter-examples.

In the following, we show the most interesting experimental results and discuss the differences in the behavior of the algorithms. We display the results of the two groups of experiments in Figures 9.6 and 9.8, respectively.

To start with, the main results of the first group of experiments, where the algorithms are compared on different alphabet sizes, appear in Figure 9.6. The figure shows, the performance of all four algorithms with respect to the total number of membership and equivalence queries, the number of states conjectured in the final

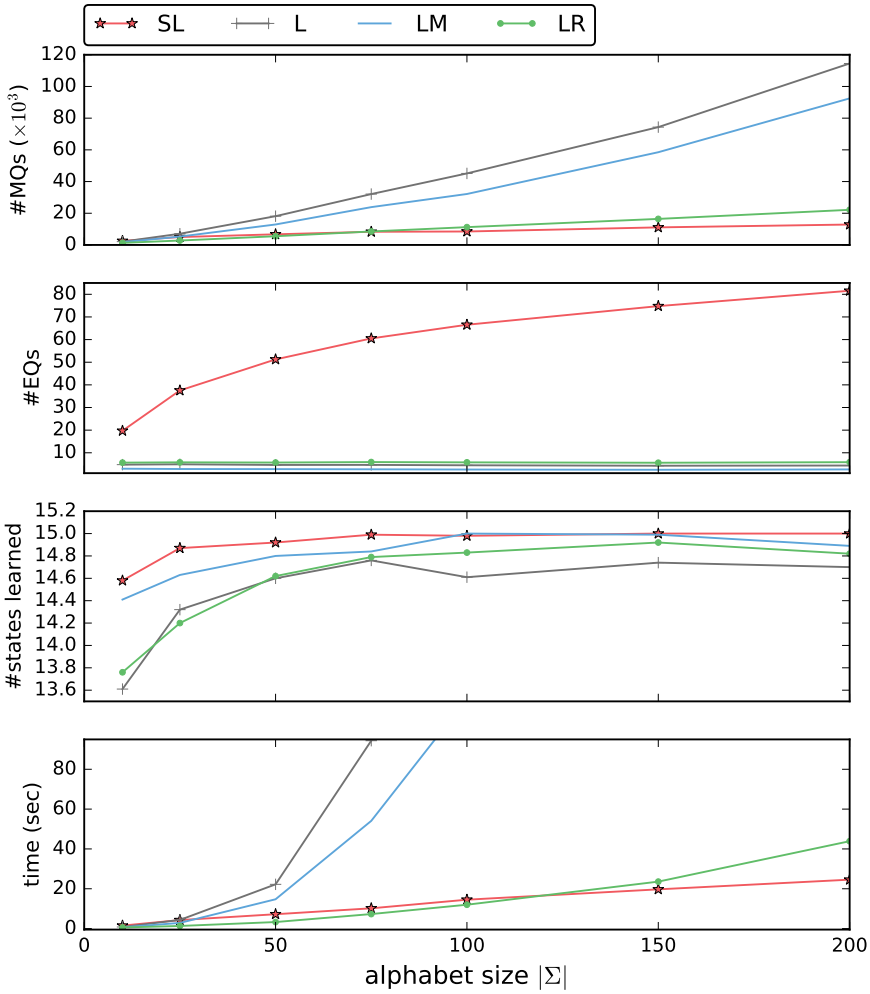


Figure 9.6: A comparison on the average number of membership queries posed, the number of equivalence checks, the states learned and the execution time for each of the algorithms: *SL* (the symbolic algorithm), *L* ([Ang87]), *LM* ([MP95]), and *LR* ([RS93]). The four algorithms are compared over several concrete alphabet sizes that ranges from 10 to 200 letters.

hypothesis, and the average execution times.

As expected, the symbolic algorithm admits the most modest growth in the number of membership queries, shown in the first row of Figure 9.6. It outperforms, in this sense, the best of the concrete algorithms, that is, the reduced algorithm (LR). This becomes more evident when the alphabet size grows beyond a certain point. We present the same results in Figure 9.7, where now the num-

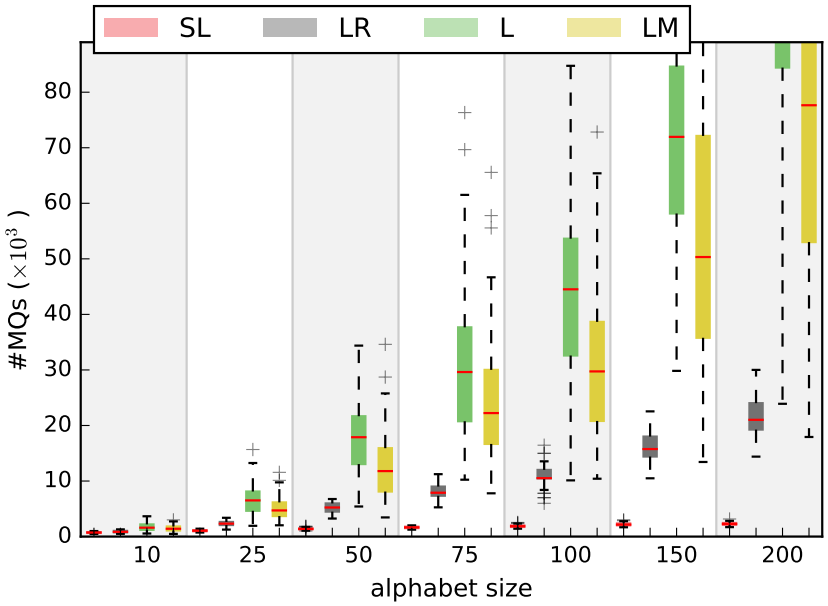


Figure 9.7: A box plot of the number of MQ's posed by each algorithm. This plot representation provides the average, variance, range and error points for each algorithm. Each column represents the experimental results where the size of the alphabet is kept fixed to sizes $10, 25, 80, 75, 100, 150$ and 200. One can see that, especially when the size of the alphabet is growing, even the (SL) worst case outperforms the best values of all other algorithms.

ber of membership queries is shown in the form box-plots. This representation provides a more clear visualization that contains the standard deviations and min max values in the data, rather than just the average. We observe that the behavior of the symbolic algorithm is better than that of the other algorithms, but what is more interesting is the fact that even the worst case run of the symbolic algorithm outperforms the best run of all other algorithms.

Referring to the number of states conjectured in the final hypothesis, shown in the third plot of Figure 9.6, we observe that the symbolic algorithm (SL) almost always discovers more states than the other algorithms. Taking into account that all algorithms produce automata that are minimal in the number of states, we can say that SL comes closer to the target automaton representation than the other algorithms. This seems to be a consequence of the increased number of equivalence queries that the algorithm asks in contrast to the other algorithms (see the second row in Figure 9.6).

Surprisingly, in terms of execution times, see last plot of Figure 9.6, the concrete reduced L^* algorithm (LR) has a similar performance to the symbolic algorithm (SL) when the alphabet size is smaller than 150, a result that is probably due

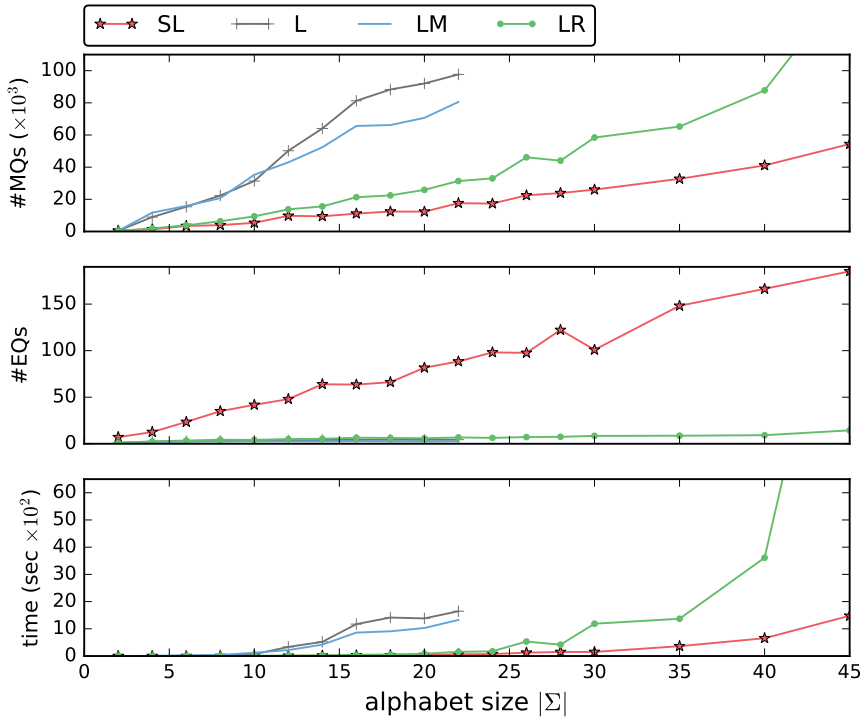


Figure 9.8: A comparison of the average number of membership queries posed, the number of equivalence checks, and the execution time for each of the algorithms: *SL* (the symbolic algorithm), *L* ([Ang87]), *LM* ([MP95]), and *LR* ([RS93]). The four algorithms are compared on variable sizes of target automata where the number of states ranges from 3 to 45.

to the increased number of equivalence queries that the symbolic algorithm needs.

Figure 9.8 shows the results of the second group of experiments. Here the algorithms are compared on different sizes of target automata, where the number of states ranges from 0 to 45. Similarly to the first group of experiments, one can see that beyond 20 states the symbolic algorithm is better than all the non-symbolic algorithms in terms of membership queries, see the first row. The number of equivalence queries, naturally larger for the symbolic algorithm, increases linearly in the number of states in the automaton.

Finally, referring to execution timings, the concrete reduced L^* algorithm (LR) has a similar performance to the symbolic algorithm (SL) only when the size of the target automata is kept smaller than 25 states.

To conclude this comparison, the symbolic algorithm asks much fewer membership queries in order to fill the table than any of the concrete algorithms, which need to ask queries for every $a \in \Sigma$. On the other hand, the symbolic algorithm gets more counter-examples, generates more hypotheses and needs more equiva-

0	NUL	16	DLE	32	SPC	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Figure 9.9: The table of all ascci characters, split into groups of *control characters*, *alphabetic*, *numerals*, and *punctuation symbols*.

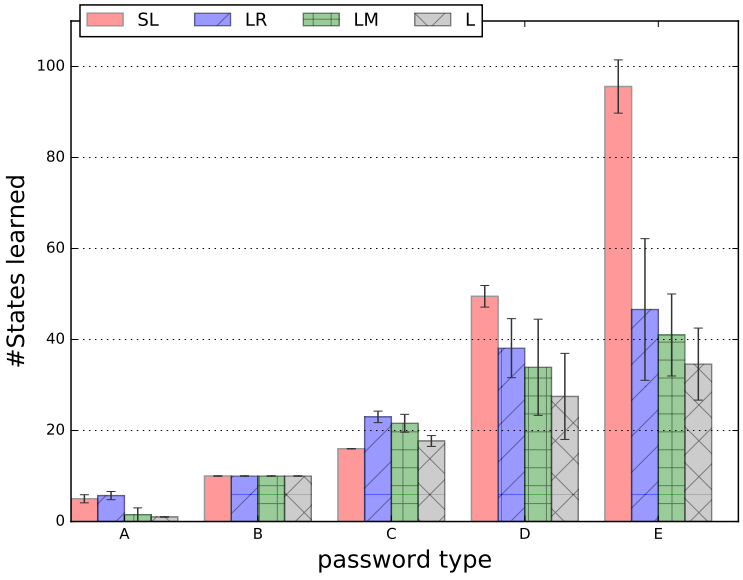
lence queries. Remarkably, this does not affect the performance of the symbolic algorithm, which is still better in the total number of membership queries that are posed, including the queries used for testing.

9.4 Learning Passwords

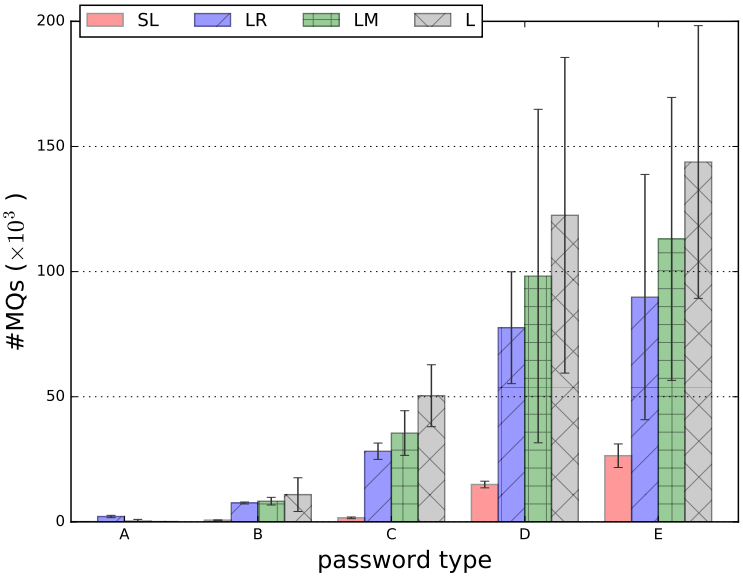
In this section, we compare the same four algorithms in a more realistic setting. We let the target language be a set of valid passwords, where the input alphabet is the set of ASCII characters, represented by the integers $\Sigma = \{0, \ldots, 127\}$. A password is a string over Σ, and it is characterized as valid when certain rules are satisfied. Rules of this kind typically concern the length of a password and groups of characters that it may or may not contain.

The ASCII characters are split into groups, see Figure 9.9. Namely, *control characters*, i.e., $\{0, \ldots, 31\} \cup \{127\} \subseteq \Sigma$, which are non printable characters that provide meta-information; *alphabetic* of two types, upper case and lower case letters, that is, $\{65, \ldots, 90\}$ and $\{97, \ldots, 122\}$ respectively; *numerals*, i.e., $\{48, \ldots, 57\}$; and, finally, the *punctuation symbols*, that is the set which contains all remaining characters.

Passwords are used in several types of accounts and often providers set up rules on how passwords should be formulated in order to guaranty security and/or simplicity to the users. Due to this, we are familiar with PIN codes, typically used for phones, credit cards, etc., but also longer passwords that should contain letters and/or punctuation characters, most commonly used for online accounts such as email accounts, bank accounts, etc. Passwords with characters from more than one



(a)



(b)

Figure 9.10: A performance comparison of the symbolic algorithm (SL) to the three concrete algorithms (LR), (LM) and (L). The four algorithms are evaluated on learning a set of different types of valid passwords: (A) PIN code, (B) easy password, (C) medium password, (D) medium-strong password, and (E) strong password. The average number of states learned by each algorithm for each password type is shown in (a), while (b) illustrates the average number of membership queries used in total.

or two groups of ASCII characters are considered stronger and more secure.

In the sequel, we define five different target languages, each one accepting a different type of valid passwords. Each language is defined by a set of rules, which are of increasing complexity resulting in languages of varying difficulty degree. For all languages selected here, no password is valid when it contains any non-printable character.

The PIN code passwords (A) contains 4 to 8 numbers. The *easy* passwords (B), with a maximum length of 8, contain any printable ASCII character, i.e., numerical, letter, or punctuation character. The *medium* passwords (C) contain from 6 up to 14 characters and use at least one character from two different sub-alphabets of printable characters; punctuation symbols are not allowed. The *medium-strong* passwords (D), with the same length as the medium passwords, require the presence of both lower-case letters and numerals and punctuation characters are allowed. Finally, the *strong* passwords (E), contain at least one character from all different sub-alphabets.

After applying all four algorithms to the different target languages, we obtain the results of 10 runs for each algorithm. A visual comparison of the results is summarized in Figure 9.10, where one can observe that the symbolic algorithm requires fewer MQ's than any other method in all five password types, see Figure 9.10b. The difference increases as the target becomes more complicated and requires more states and transitions. Remarkable is the fact that the symbolic algorithm manages to discover more states, in almost all different password types, and especially when the target language becomes more complicated and is represented by a larger automaton, see Figure 9.10a. In Figure 9.11, we provide two instances of symbolic automata for the passwords of type (D) and (E), which are inferred using the symbolic algorithm.

9.5 Learning over the Booleans

In Chapter 7, the symbolic algorithm has been adapted to the case where the input alphabet is a set of Boolean vectors, i.e., $\Sigma = \mathbb{B}^n$. We have implemented the learning algorithm following the algorithms and procedures presented in Section 7.3. To represent the semantics of the symbols, we use binary decision trees (see Section 2.2), where, in order to learn the partitions at each state, we use an optimized version of the CART algorithm (see Section 2.3), which is available in the *scikit-learn* module of machine learning in Python [PVG+11]. As a splitting quality measure we use entropy, and best split is chosen. As a termination criterion we use purity, that is, trees are expanded until all leaves are pure and thus the observation table is evidence compatible.

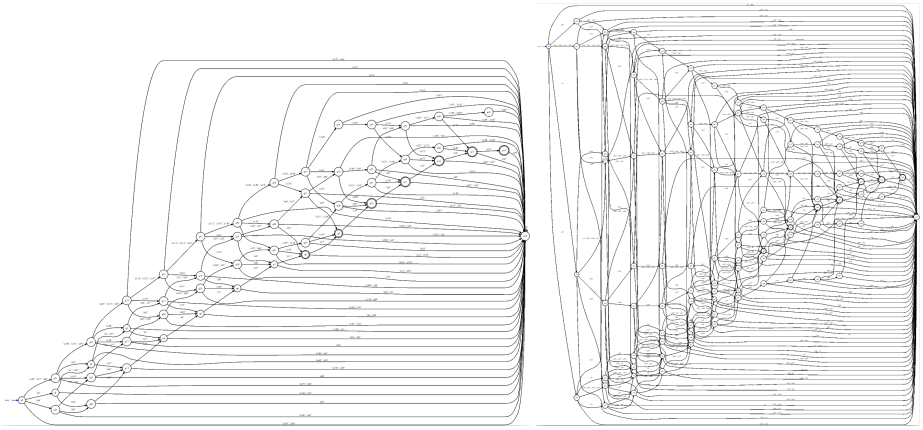


Figure 9.11: Two instances of symbolic automata for the passwords of type (D) and (E), which are learned using the symbolic algorithm.

We test the learning algorithm on two groups of experiments, as in the case of numerical alphabets in Section 9.3. The two groups of experiments are used to evaluate the algorithm's performance when the size of the automaton or the size of the alphabet increases, while other parameters are kept constant. All results are averaged over the total number of experiments for each case, that is, 10 different target automata for each automaton or alphabet size, where the learning algorithm is applied 10 times to the same target.

For the first group of experiments, we use the same fixed minimal automaton structure that appears in Figure 9.5. Partitions are randomly chosen at each state and the alphabet's size ranges from 8 to 32768 letters, that is, 3 to 15 Boolean variables. In the second group of experiments, we fix the alphabet to $\Sigma = \mathbb{B}^8$ and generate random automata of increasing size, with the number of states ranging from 2 to 50. Transitions are determined by randomly generated pseudo-boolean functions. They are limited to contain at most $k = 4$ literals per term, which, intuitively, restricts to BDTs of maximal depth k and corresponds to at most 2^k outgoing transitions per state.

As in Section 9.3, we have measured several size and time parameters. Size related, we consider the size of the final observation table, the concrete and symbolic sample size, as well as the number of membership and equivalence queries posed to the teacher.

The major observation of our experimental results for the Boolean alphabets show that, if the complexity of the alphabet partition remains fixed, the number of membership, equivalence queries and sample size remain constant as the alphabet increases. This can be seen in Figure 9.12. Note that, the number of variables increases, but the depth of the trees that are used to define the partitions are bounded

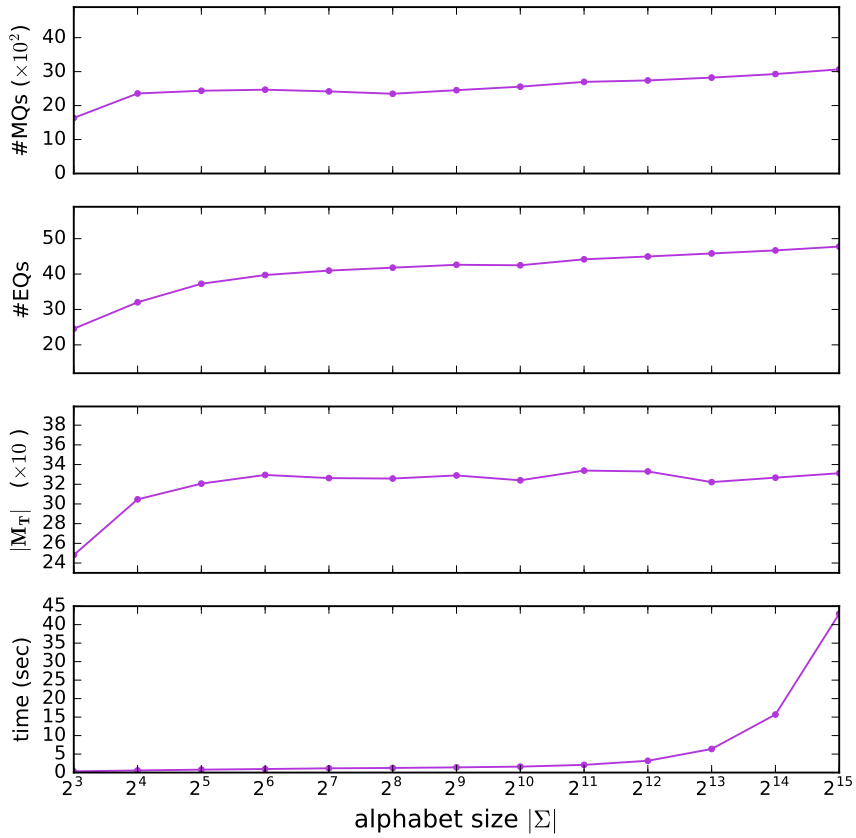


Figure 9.12: The performance of the symbolic algorithm when applied on the Boolean vector alphabet $\Sigma = \mathbb{B}^n$ where the number of variables n ranges from 3 to 15. Here we illustrate the average number of membership queries, equivalence queries, the size of the final sample, and the execution time as the alphabet size increases.

to have at most depth 4. That is, the variables that play a crucial role in defining a partition at each state is limited to 4.

A growth in the total execution time seems exponential in the number of variables, which corresponds to linear growth in the alphabet size, see the last row of Figure 9.12. Profiling shows that most of the time is spent in testing, indicating that, in the implementation, this depends on the size of the concrete alphabet.

On the other hand, as we see in Figure 9.13, the number of queries, the size of the symbolic sample, the number of hypotheses conjectured, and the total execution time all appear to increase linearly in the number of states in the target automaton. This result does not come as a surprise since the number of BDTs that need to be

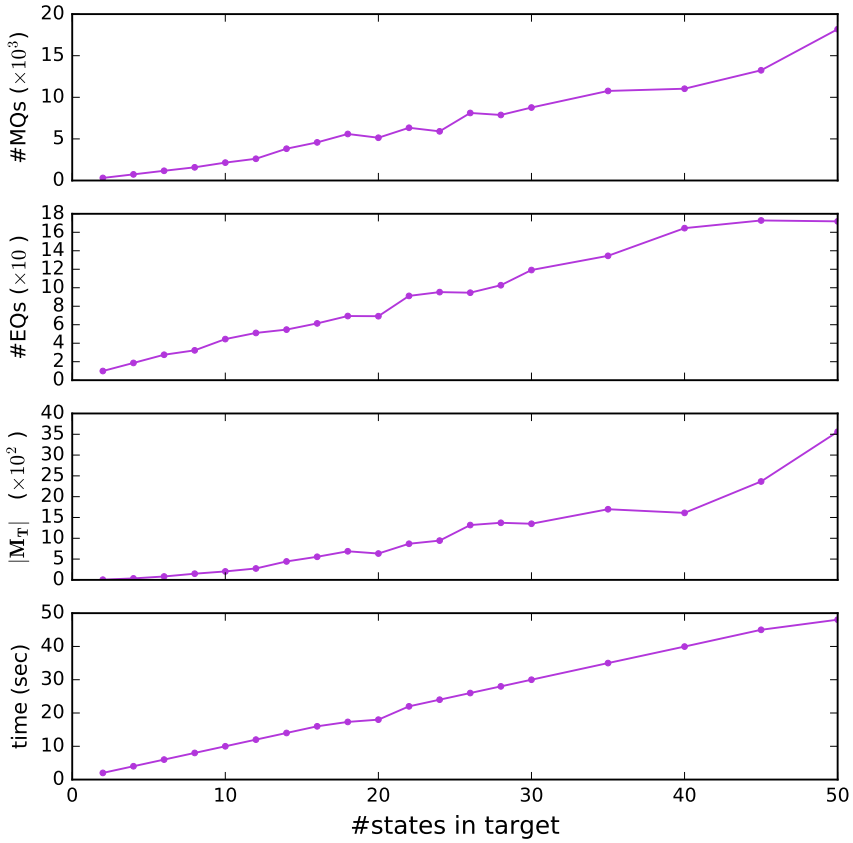


Figure 9.13: The performance of the symbolic algorithm when applied on target automata of variable size, where the number of states in the minimal automaton that represents the target ranges from 3 to 50. Here we illustrate the average number of membership queries, equivalence queries, the size of the final sample, and the execution time as the size of the automaton increases.

learned depends on the number of states.

9.6 Comparing Boolean Vectors to Numerical Alphabets

Comparison over Alphabet and Target Size

In this section, we compare the results that appear in Sections 9.3 and 9.5, which are related to the two symbolic algorithms, i.e., the symbolic learning algorithm on numbers (SL) and on the Booleans (SL Booleans), respectively. All results are illustrated in Figure 9.14.

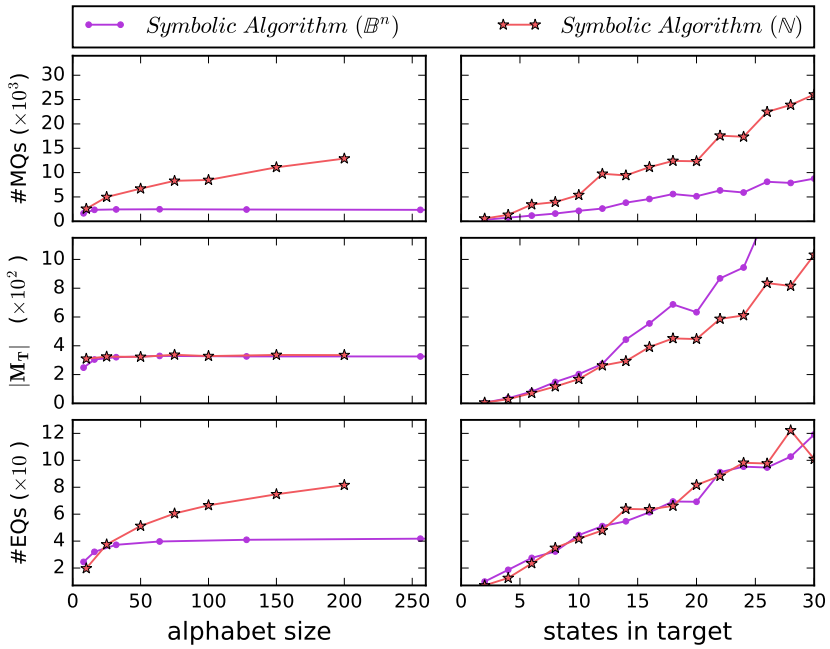


Figure 9.14: A comparison on the performance of the symbolic algorithm when applied on interval alphabets (SL) and Boolean vector alphabets (SL Booleans). The rows show 1) the average number of membership queries, 2) the average size of the symbolic sample, and 3) the average number of equivalence queries posed in total by each algorithm, respectively.

In the first group of experiments (first column), the same fixed underlying structure is used to generate the target languages while the alphabet size grows. We observe that the Boolean algorithm clearly outperforms the numerical algorithm in terms of membership and equivalence queries. In the second group of experiments (second column), where we compare over different automaton sizes, the numerical algorithms use a fixed alphabet of 150 letters, while the Boolean algorithm uses a 256-letter alphabet. Although the absolute numbers differ for the three algorithms, it is worth noting that the growth of the number of queries, as well as the size of the symbolic sample is linear with respect to the size of the automaton in both symbolic algorithms. We observe again that in the case of Boolean vectors, the number of membership queries is much smaller than for numerical alphabets.

Finally, in the last plot of the first group of experiments (first column, third row of Figure 9.14), we can see that fewer EQ's are needed in general for the case of Boolean vector alphabets. A possible explanation for this superiority can rely on the different restrictions that were applied in defining the partitions in the target languages. To explain this better, we give some insights on the random generator that is used to define the target languages. For interval partitions, a restriction to

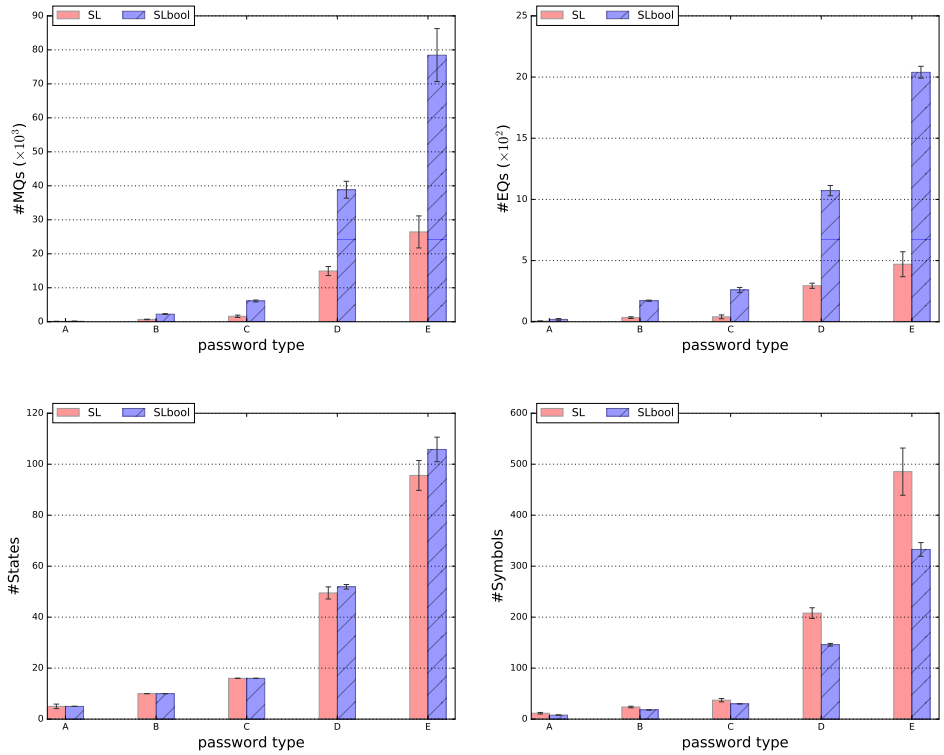


Figure 9.15: The performance of the symbolic algorithm on learning languages that represent valid passwords. Two adaptations of the algorithm are compared on two different representations of the input alphabet, the set of ASCII characters. In the first, the alphabet is a set of integers (SL), and in the second, it is a set of Boolean vectors (SLBool).

k intervals implies that their size itself grows linearly with the alphabet size and thus one may expect more errors and counter-examples when the boundaries are approximated. This is often the case, especially when some intervals are small compared to others. On the other hand, for partitions on the Boolean cube, the decision trees are restricted to have at most depth k. This, in turn, restricts the number of variables that interact in a product alphabet, and it gives a lower-bound on the size of the Boolean sub-cubes appearing in the partition. This lower bound grows linearly with the size of the alphabet and such partitions are easier to learn.

Comparing on Learning Passwords

In Section 9.4, the symbolic algorithm is used to learn languages that represent valid passwords. The set of ASCII characters, in their decimal representation, is used as the input alphabet. The alphabet $\Sigma = \{0, \ldots, 127\}$ is partitioned into intervals, where intervals represent groups of characters, e.g., *digits*, *lowercase*

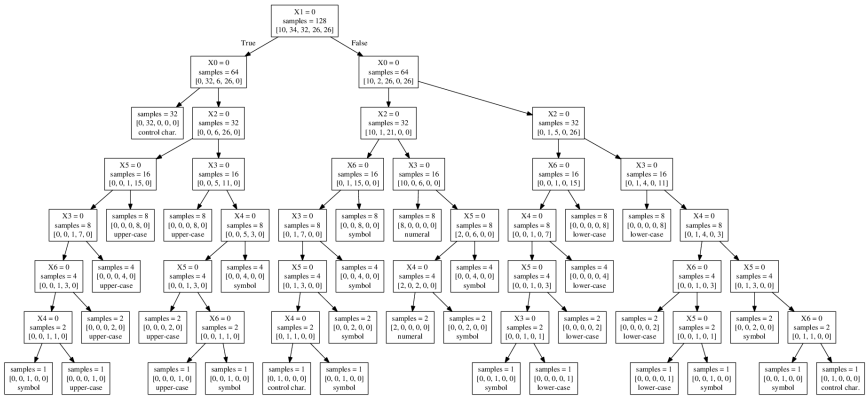


Figure 9.16: The BDT representing the partition of the ASCII characters into groups of characters, i.e., *digits*, *lowercase letters*, *punctuation symbols*, etc.

letters, *special symbols*, etc.

In this section, we apply the symbolic algorithm on the same set of target languages, but this time, we use as an input alphabet the set $\Sigma = \mathbb{B}^7$ of Boolean vectors, which corresponds to the set of ASCII characters' binary representation. For instance, the letter 'a' corresponds to the integer 97 and to the Boolean vector (1100001). In contrast to the interval partitions, which appears to be convenient when classifying ASCII into groups of characters, the Boolean vector adaptation requires a complicated and deep BDT to determine all partition blocks in the alphabet, see Figure 9.16.

Figure 9.15 shows the performance of the two algorithms. We compare them on the number of queries they use as well as the size of their final conjectures in terms of number of states and transitions. As we can observe, due to the higher complexity on the partitions for the Boolean case, the algorithm requires more evidences to define the partitions well. Hence, more equivalence and membership queries are used.

The two algorithms terminate with conjectures that contain similar numbers of states. A small advantage is observed in the case of Boolean Vectors as valid passwords become more complicated. Although, this is not surprising when we consider the increased number of equivalence queries the algorithm uses.

Finally, concerning the size of the symbolic alphabets, in the case of intervals more symbols are used. This is due to the fact, that partitions and semantics are determined in a slightly different way in the two adaptations of the algorithm. That is, BDTs use the same symbol to represent multiple leaf nodes allowing at most one transition that leads from a state q to a state q'. In contrast, in the case of the intervals, the semantics are restricted to preserve convexity, and thus, multiple transitions from q to q' are allowed.

We conclude that both algorithms can be used when the alphabet admits both representations resulting equivalent automata. However, the complexity on the partitions should be taken into account, as it affects the number of resources the algorithm uses.

9.7 Conclusions

In this chapter, we have experimented with the symbolic algorithm and applied it to several case studies. Moreover, we compared the symbolic algorithm to other known concrete algorithms, restricting the alphabet to be finite. The symbolic algorithm always terminates faster and with better results in the PAC setting. Our algorithm, almost always, discovers more states and since we consider that the resulting automaton is minimal, it means that the conjecture is close to the target's representation. The size of the sample and the number of membership queries are notably lower than any other algorithm, since the words are partially queried and evidences are used to represent larger set of letters. Naturally, this approach increases the number of equivalence queries our algorithm requires, however, this does not seem to influence the time that the algorithm needs until termination.

Comparing the symbolic algorithms on different alphabets, we observe that when applied to Boolean vectors, it performs better, partially due to the different nature of partition complexity restrictions on decision trees and intervals, and also the fact that we use a better method to learn partition boundaries. Learning BDTs allows a smaller boundary error than the simple splitting we use in the interval case. The latter can be ameliorated by using a more precise method, such as binary search, for detecting the boundaries.

10 CHAPTER
Conclusions and Future Work

In this thesis we presented an algorithmic scheme for learning languages over large alphabets. We aimed at languages accepted by symbolic automata with a modest number of states and transitions. The transitions are guarded by simple constraints on the alphabet, which by itself can be arbitrarily large. The constraints form a partition of the alphabet in each state and hence the automaton is deterministic. Our framework can be characterized by the following features:

1. It is based on a clean and general definition of the relation between the concrete and symbolic alphabets, via concepts such as alphabet partitions, evidences and representatives;
2. It separates the sequential aspects of learning (modifying the automaton structure) and the learning of partition boundaries, which is specific to the alphabet;
3. It treats the modification of alphabet partitions in a rigorous way based on the concept of evidence compatibility. This guarantees that no superfluous symbols are introduced;
4. It can work with or without a helpful teacher, where in the latter case, equivalence checking is done by random sampling and exact learnability is replaced by PAC;
5. It is modular, separating the generic aspects from those that are alphabet specific, thus providing for a relatively easy adaptation to new alphabets.

We have implemented the testing-based algorithm for two classes of alphabets, numerical (sub-intervals of $\mathbb{R}$ or $\mathbb{N}$) and Boolean ($\mathbb{B}^n$). For the former, symbolic letters were associated with sub-intervals while in the latter we used decision trees to define the alphabet partitions in every state. For both cases we ran some initial experiments on mostly synthetic examples, to see how the empirical performance

of the algorithm varies with the size of the automaton and the alphabets. The preliminary results were encouraging and the symbolic algorithm outperforms concrete learning algorithms over the same finite alphabets. There is still much to do in tuning and optimizing our algorithm.

What is still missing is a convincing class of real-world applications that benefits from such algorithms. In the numerical domain, these should be mechanisms where discrete transitions are taken based on threshold crossings of numerical variables without remembering their precise values. In the Boolean domain, one needs to find applications in the formal specification of large complex systems with many components (digital circuit, distributed multi-agent systems). Hopefully such specifications could be expressible by symbolic automata where the complexity can be confined to the alphabet partitions while the number of states does not explode.

A natural future extension of the algorithm is to consider alphabets which are subsets of $\mathbb{N}^n$ and $\mathbb{R}^n$. Preliminary work in this direction has been reported in Section 6.2, but it used a very restricted type of monotone partitions in order to keep the notion of a minimal counter-example meaningful. In such a future algorithm one can use more general partitions, represented by regression trees, a generalization of decision trees to numerical domains.

We feel that the application of our framework for learning over Boolean alphabets is very promising and is worth being explored much further. It combines well-known concepts and algorithms from traditional machine learning (decision trees) with automaton learning. As such, it can suggest an alternative to other machine learning approaches (recurrent neural learning, decision tree learning over variables that correspond to different time indices, auto-correlation) used to learn from data-rich and time-dependent examples. Automata bring some sophistication and economy in representing sets of time series and might bring a genuine practical contribution to machine learning. This will require, however, some changes in the automaton learning approach, most notably to allow noise in the examples and to relax the active learning framework where the learner can fully dictate the set of observed examples.

Bibliography

[ABD15] Eugene Asarin, Nicolas Basset, and Aldric Degorre. Entropy of regular timed languages. *Information and Computation*, 241:142–176, 2015. (*Cited on page 86.*)

[AJU10] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *IFIP International Conference on Testing Software and Systems*, pages 188–204. Springer, 2010. (*Cited on page 4.*)

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. (*Cited on pages 1, 21, 23, 28, 47, 90, 97, 99, and 101.*)

[Ang88] Dana Angluin. Queries and concept learning. *Machine learning*, 2(4):319–342, 1988. (*Cited on page 90.*)

[BB13] Matko Botinčan and Domagoj Babić. Sigma*: Symbolic learning of input-output specifications. In *POPL*, pages 443–456. ACM, 2013. (*Cited on page 48.*)

[BFSO84] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984. (*Cited on pages 16, 17, 20, and 78.*)

[BJR06] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *FASE*, volume 3922 of *LNCS*, pages 107–121. Springer, 2006. (*Cited on page 48.*)

[BKR96] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *International Workshop on Implementing Automata*, pages 6–25. Springer, 1996. (*Cited on page 33.*)

[BLP10] Michael Benedikt, Clemens Ley, and Gabriele Puppis. What you must remember when processing data words. In *AMW*, volume 619 of *CEUR Workshop Proceedings*, 2010. (*Cited on page 47.*)

[BMS+06] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 7–16. IEEE, 2006. (*Cited on page 33.*)

[BR05] Therese Berg and Harald Raffelt. 19 Model checking. In *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *LNCS*, pages 557–603. Springer, 2005. (*Cited on pages 2, 21, and 29.*)

[Brz62] Janusz A Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical theory of Automata*, 12(6):529–561, 1962. (*Cited on page 10.*)

[CHJS16] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active learning for extended finite state machines. *Formal Aspects of Computing*, 28(2):233–263, 2016. (*Cited on page 47.*)

[CSS+10] Chia Yuan Cho, Eui Chul Richard Shin, Dawn Song, et al. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 426–439. ACM, 2010. (*Cited on page 4.*)

[DD17] Samuel Drews and Loris D'Antoni. Learning symbolic automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–189. Springer, 2017. (*Cited on page 48.*)

[DlH10] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010. (*Cited on pages 21 and 22.*)

[DV14] Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. In *POPL*, pages 541–554. ACM, 2014. (*Cited on pages 34, 36, 38, and 39.*)

[E88] Utgoff Paul E. ID5: an incremental ID3. In John E. Laird, editor, *Machine Learning, Proceedings of the Fifth International Conference on Machine Learning, Ann Arbor, Michigan, USA, June 12-14,*

1988, pages 107–120. Morgan Kaufmann, 1988. (*Cited on pages 16 and 19.*)

[GKS10] Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. Variable automata over infinite alphabets. In *International Conference on Language and Automata Theory and Applications*, pages 561–572. Springer, 2010. (*Cited on page 33.*)

[Gol67] E Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967. (*Cited on page 23.*)

[Gol72] E Mark Gold. System identification via state characterization. *Automatica*, 8(5):621–636, 1972. (*Cited on pages 10 and 23.*)

[Gol78] E Mark Gold. Complexity of automaton identification from given data. *Information and control*, 37(3):302–320, 1978. (*Cited on page 23.*)

[HJJ+95] Jesper G Henriksen, Ole JL Jensen, Michael E Jrgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders B Sandholm. Mona: Monadic second-order logic in practice. In *TACAS*, volume 1019 of *LNCS*, pages 80–110. Springer, 1995. (*Cited on page 33.*)

[HMU06] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. (*Cited on pages 7 and 37.*)

[Hop71] John E Hopcroft. An $n\ logn$ algorithm for minimizing states in a finite automaton. Technical report, DTIC Document, 1971. (*Cited on page 10.*)

[HSJC12] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In *VMCAI*, volume 7148 of *LNCS*, pages 251–266. Springer, 2012. (*Cited on page 47.*)

[HSM11] Falk Howar, Bernhard Steffen, and Maik Merten. Automata learning with automated alphabet abstraction refinement. In *VMCAI*, volume 6538 of *LNCS*, pages 263–277. Springer, 2011. (*Cited on page 48.*)

[HV11] Pieter Hooimeijer and Margus Veanes. An evaluation of automata algorithms for string analysis. In *VMCAI*, volume 6538 of *LNCS*, pages 248–262. Springer, 2011. (*Cited on pages 34, 36, and 37.*)

[IHS13] Malte Isberner, Falk Howar, and Bernhard Steffen. Inferring automata with state-local alphabet abstractions. In *NASA Formal Methods*, volume 7871 of *LNCS*, pages 124–138. Springer, 2013. (*Cited on page 48.*)

[IS14] Malte Isberner and Bernhard Steffen. An abstract framework for counterexample analysis in active automata learning. In *ICGI*, pages 79–93, 2014. (*Cited on page 29.*)

[KF70] Andreĭ Nikolaevich Kolmogorov and Sergeĭ Vasilevich Fomin. *Introductory real analysis*. New York: Dover Publications Inc., 1970. (*Cited on page 10.*)

[KF94] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994. (*Cited on pages 33 and 47.*)

[LGJ07] Tristan Le Gall and Bertrand Jeannet. Lattice automata: A representation for languages on infinite alphabets, and some applications to verification. In *International Static Analysis Symposium*, pages 52–68. Springer, 2007. (*Cited on page 33.*)

[LP97] Harry R Lewis and Christos H Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, 1997. (*Cited on page 7.*)

[MM14] Oded Maler and Irini-Eleftheria Mens. Learning regular languages over large alphabets. In *TACAS*, volume 8413 of *LNCS*, pages 485–499. Springer, 2014. (*Cited on page 5.*)

[MM15] Oded Maler and Irini-Eleftheria Mens. Learning regular languages over large ordered alphabets. *Logical Methods in Computer Science (LMCS)*, 11(3), 2015. (*Cited on page 5.*)

[MM17] Oded Maler and Irini-Eleftheria Mens. A generic algorithm for learning symbolic automata from membership queries. In *Models, Algorithms, Logics and Tools*, pages 146–169. Springer, 2017. (*Cited on page 5.*)

[MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004. (*Cited on page 3.*)

[MN10] Karl Meinke and Fei Niu. A learning-based approach to unit testing of numerical software. In *IFIP International Conference on Testing Software and Systems*, pages 221–235. Springer, 2010. (*Cited on page 4.*)

[MNP08] Oded Maler, Dejan Nickovic, and Amir Pnueli. Checking temporal properties of discrete, timed and continuous behaviors. In *Pillars of computer science*, pages 475–505. Springer, 2008. (*Cited on page 3.*)

[Moo56] Edward F Moore. Gedanken-experiments on sequential machines. In *Automata studies*, volume 34 of *Annals of Mathematical Studies*, pages 129–153. Princeton, 1956. (*Cited on pages 10 and 23.*)

[MP95] Oded Maler and Amir Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316–326, 1995. (*Cited on pages 27, 28, 97, 99, and 101.*)

[Mur12] Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 1 edition, 2012. (*Cited on page 17.*)

[Ner58] Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. (*Cited on pages 1, 9, and 23.*)

[Nie03] Oliver Niese. *An integrated approach to testing complex systems*. PhD thesis, Universität Dortmund, 2003. (*Cited on page 4.*)

[NSV04] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic (TOCL)*, 5(3):403–435, 2004. (*Cited on page 33.*)

[OG92] Jose Oncina and Pedro Garcia. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition*, volume volume 5 of *Series in Machine Perception and Artificial Intelligence*, pages 99–108. World Scientific, 1992. (*Cited on page 23.*)

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. (*Cited on page 104.*)

[PW93] Leonard Pitt and Manfred K Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *Journal of the ACM (JACM)*, 40(1):95–142, 1993. (*Cited on page 23.*)

[Qui86] J Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986. (*Cited on pages 16 and 78.*)

[Qui93] J Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. (*Cited on pages 16 and 20.*)

[RMSM09] Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. *International journal on software tools for technology transfer*, 11(4):307–324, 2009. (*Cited on page 4.*)

[RS93] Ronald L Rivest and Robert E Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993. (*Cited on pages 27, 29, 97, 99, and 101.*)

[Seg06] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *International Workshop on Computer Science Logic*, pages 41–57. Springer, 2006. (*Cited on page 33.*)

[SF86] Jeffrey C Schlimmer and Douglas Fisher. A case study of incremental concept induction. In *AAAI*, pages 496–501, 1986. (*Cited on pages 16 and 19.*)

[Sha08] Muzammil Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Components to support Integration Testing*. PhD thesis, Grenoble Institute of Technology, 2008. (*Cited on page 4.*)

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006. (*Cited on page 7.*)

[SL07] Guoqiang Shu and David Lee. Testing security properties of protocol implementations-a machine learning based approach. In *27th International Conference on Distributed Computing Systems (ICDCS'07)*, pages 25–25. IEEE, 2007. (*Cited on page 4.*)

[SVVV14] Rick Smetsers, Michele Volpato, Frits Vaandrager, and Sicco Verwer. Bigger is not always better: on the quality of hypotheses in active automata learning. In *International Conference on Grammatical Inference*, pages 167–181, 2014. (*Cited on page 85.*)

[UBC97] Paul E Utgoff, Neil C Berkman, and Jeffery A Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1):5–44, 1997. (*Cited on page 19.*)

[Utg89] Paul E Utgoff. Incremental induction of decision trees. *Machine learning*, 4(2):161–186, 1989. (*Cited on pages 16, 19, 20, and 78.*)

[Val84] Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984. (*Cited on page 90.*)

[VBDM10] Margus Veanes, Nikolaj Bjørner, and Leonardo De Moura. Symbolic automata constraint solving. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, volume 6397, pages 640–654. Springer, 2010. (*Cited on pages 4 and 34.*)

[VDHT10] Margus Veanes, Peli De Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. pages 498–507, 2010. (*Cited on pages 4, 33, 34, and 37.*)

[Vea13] Margus Veanes. Applications of symbolic finite automata. In *International Conference on Implementation and Application of Automata*, volume 7982, pages 16–23. Springer, 2013. (*Cited on pages 4, 33, and 34.*)

[VGDHT09] Margus Veanes, Pavel Grigorenko, Peli De Halleux, and Nikolai Tillmann. Symbolic query exploration. In *International Conference on Formal Engineering Methods*, pages 49–68. Springer, 2009. (*Cited on page 34.*)

[VHL+12] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Björner. Symbolic finite state transducers: algorithms and applications. In *POPL*, pages 137–150. ACM, 2012. (*Cited on page 34.*)

[VNG01] Gertjan Van Noord and Dale Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4(3):263–286, 2001. (*Cited on pages 4, 33, 34, 36, and 37.*)

[VTDH10] Margus Veanes, Nikolai Tillmann, and Jonathan De Halleux. Qex: Symbolic SQL query explorer. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 425–446. Springer, 2010. (*Cited on page 34.*)

[Wat96] Bruce W Watson. Implementing and using finite automata toolkits. *Natural Language Engineering*, 2(04):295–302, 1996. (*Cited on page 33.*)

[WBAH08] Neil Walkinshaw, Kirill Bogdanov, Shaukat Ali, and Mike Holcombe. Automated discovery of state transitions and their functions in source code. *Software Testing, Verification and Reliability*, 18(2):99–121, 2008. (*Cited on page 4.*)

[YBCI08] Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H Ibarra. Symbolic string verification: An automata-based approach. In *International SPIN Workshop on Model Checking of Software*, pages 306–324. Springer, 2008. (*Cited on pages 33 and 34.*)

Learning Regular Languages over Large Alphabets

Irini-Eleftheria Mens

Thèse dirigée par **Oded Maler**

Learning regular languages is a branch of machine learning, a domain which has proved useful in many areas, including artificial intelligence, neural networks, data mining, verification, etc. In addition, interest in languages defined over large and infinite alphabets has increased in recent years. Although many theories and properties generalize well from the finite case, learning such languages is not an easy task. As the existing methods for learning regular languages depend on the size of the alphabet, a straightforward generalization in this context is not possible.

In this thesis, we present a generic algorithmic scheme that can be used for learning languages defined over large or infinite alphabets, such as bounded subsets of $\mathbb{N}$ or $\mathbb{R}$ or Boolean vectors of high dimensions. We restrict ourselves to the class of languages accepted by deterministic symbolic automata that use predicates to label transitions, forming a finite partition of the alphabet for every state.

Our learning algorithm, an adaptation of Angluin's L^*, combines standard automaton learning by state characterization, with the learning of the static predicates that define the alphabet partitions. We use the online learning scheme, where two types of queries provide the necessary information about the target language. The first type, membership queries, answer whether a given word belongs or not to the target. The second, equivalence queries, check whether a conjectured automaton accepts the target language and provide a counter-example otherwise.

We study language learning over large or infinite alphabets within a general framework but our aim is to provide solutions for particular concrete instances. For this, we focus on the two main aspects of the problem. Initially, we assume that equivalence queries always provide a counter-example which is minimal in the length-lexicographic order when the conjecture automaton is incorrect. Then, we drop this "strong" equivalence oracle and replace it by a more realistic assumption, where equivalence is approximated by testing queries, which use sampling on the set of words. Such queries are not guaranteed to find counter-examples and certainly not minimal ones. In this case, we obtain the weaker notion of PAC (probably approximately correct) learnability and learn an approximation of the target language. All proposed algorithms have been implemented and their performance, as a function of automaton and alphabet size, has been empirically evaluated.

L'apprentissage de langages réguliers est une branche de l'apprentissage automatique qui s'est révélée utile dans de nombreux domaines tels que l'intelli-gence artificielle, les réseaux de neurones, l'exploration de données, la vérification, etc. De plus, l'intérêt dans les langages définis sur des alphabets infinis ou de grande taille s'est accru au fil des années. Même si plusieurs propriétés et théories se généralisent à partir du cas fini, l'apprentissage de tels langages est une tâche difficile. En effet, dans ce contexte, l'application naïve des algorithmes d'apprentissage traditionnel n'est pas possible.

Dans cette thèse, nous présentons un schéma algorithmique général pour l'ap-prentissage de langages définis sur des alphabets infinis ou de grande taille, comme par exemple des sous-ensembles bornés de $\mathbb{N}$ or $\mathbb{R}$ ou des vecteurs booléens de grandes dimensions. Nous nous restreignons aux classes de langages qui sont acceptés par des automates déterministes symboliques utilisant des prédicats pour définir les transitions, construisant ainsi une partition finie de l'alphabet pour chaque état.

Notre algorithme d'apprentissage, qui est une adaptation du L^* d'Angluin, combine l'apprentissage classique d'un automate par la caractérisation de ses états, avec l'apprentissage de prédicats statiques définissant les partitions de l'alphabet. Nous utilisons l'apprentissage incrémental avec la propriété que deux types de requêtes fournissent une information suffisante sur le langage cible. Les requêtes du premier type sont les requêtes d'appartenance, qui permettent de savoir si un mot proposé appartient ou non au langage cible. Les requêtes du second type sont les requêtes d'équivalence, qui vérifient si un automate proposé accepte le langage cible; dans le cas contraire, un contre-exemple est renvoyé.

Nous étudions l'apprentissage de langages définis sur des alphabets infinis ou de grande tailles dans un cadre théorique et général, mais notre objectif est de proposer des solutions concrètes pour un certain nombre de cas particuliers. Ensuite, nous nous intéressons aux deux principaux aspects du problème. Dans un premier temps, nous supposerons que les requêtes d'équivalence renvoient toujours un contre-exemple minimal pour un ordre de longueur-lexicographique quand l'automate proposé est incorrect. Puis dans un second temps, nous relâchons cette hypothèse forte d'un oracle d'équivalence, et nous la remplaçons avec une hypothèse plus réaliste où l'équivalence est approchée par un test sur les requêtes qui utilisent un échantillonnage sur l'ensemble des mots. Dans ce dernier cas, ce type de requêtes ne garantit pas l'obtention de contre-exemples, et par conséquent de contre-exemples minimaux. Nous obtenons alors une notion plus faible d'apprent-issage PAC (*Probably Approximately Correct*), permettant l'apprentissage d'une approximation du langage cible. Tout les algorithmes ont été implémentés, et leurs performances, en terme de construction d'automate et de taille d'alphabet, ont été évaluées empiriquement.

Communauté UNIVERSITÉ Grenoble Alpes

Verimag