國立臺灣大學電機資訊學院電機工程學研究所

碩士論文

Graduate Institute of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

多樣性自動機之學習演算法與其量化分析

Quantitative Analysis using Multiplicity Automata Learning

洪文起

Wen-Chi, Hung

指導教授：王凡 博士

Advisor: Farn Wang, Ph.D.

中華民國 106 年 7 月

July, 2017

doi:10.6342/NTU201701657

國立臺灣大學碩士學位論文
口試委員會審定書

多樣性自動機之學習演算法與其量化分析
Quantitative Analysis using Multiplicity Automata Learning

本論文係洪文起君（學號 R04921051）在國立臺灣大學電機工程學系完成之碩士學位論文，於民國 106 年 07 月 13 日承下列考試委員審查通過及口試及格，特此證明。

口試委員：________________（簽名）
（指導教授）

雷欽隆 陳銘憲
陳郁方 江介宏

系主任 劉宗文 （簽名）

doi:10.6342/NTU201701657

誌謝

獻上最誠摯的感謝，給我的雙親與蘊軒、你們的支持是我一路上的精神糧食與動力。

兩年研究所生活中，我首先要感謝指導教授王凡教授、中研院陳郁方老師、王伯堯老師、蔡明憲學長，在學習過程當中，從你們身上學習到不僅僅是軟體測試領域的專業，更常蒙受許多教誨與叮嚀。

另外，要感謝雋飛、宗儒、謝橋，平日在實驗室與你們討論，總讓我受益匪淺，也感謝你們在一起修課的日子中，給予協助及幫忙。

最後，要感謝實驗室的同學們俊豪、光奇、啟文、汝瑋、睿頡、懋哲，一起修課討論作業和一起出遊的日子我真的很開心。

最後，謹以此文獻給我摯愛的雙親。

doi:10.6342/NTU201701657

中文摘要

此篇論文應用機率近似模型學習演算法學習多樣性自動機。該演算法會產生目標軟體之多樣性自動機模型，並將其應用於量化分析。使用產生之多樣性自動機模型，我們設計了一系列演算法可以預測軟體量化分析目標軟體的最大值與平均值等特性。此外，我們修改該演算法，使其在輸入字母數量並非固定時也可以通用。在此片論文中我們實際測試了五種不同類型的軟體，實驗證明我們預測的結果與暴力法算得之結果相當接近，足以證明此演算法可以產生一些傳統難以預測的數據，並具有相當高的可信度。

關鍵字：機率近似正確、機器學習、多樣性自動機、量化分析

doi:10.6342/NTU201701657

ABSTRACT

In this paper, we apply a probably approximately correct (PAC) learning algorithm for multiplicity automata which can generate a quantitative model of target system behaviors with a statistical guarantee. By using the generated multiplicity automata model, we apply two analysis algorithms to estimate the minimum, maximum and average values of system behaviors. Also, we demonstrate how to apply the learning algorithm when the alphabet symbol size is not fixed. The result of the experiment is encouraging; Our approach made the estimation which is as precise as the exact reference answer obtains by a brute force enumeration.

Keywords: probably approximately correct, machine learning, quantitative analysis, multiplicity automata

doi:10.6342/NTU201701657

CONTENTS

口試委員會審定書 .... #

誌謝 I

中文摘要 II

ABSTRACT III

CONTENTS IV

LIST OF FIGURES VII

LIST OF TABLES VIII

CHAPTER 1 INTRODUCTION 1

1.1 Motivation 1

1.2 Related Work 3

1.3 Contribution 4

CHAPTER 2 PRELIMINARIES 5

2.1 Multiplicity Automata 5

2.2 Hankel Matrix 6

CHAPTER 3 LEARNING ALGORITHM OF MA 7

doi:10.6342/NTU201701657

3.1 PAC Learning 7

CHAPTER 4 OVERVIEW 10

CHAPTER 5 ANALYZING PROPERTIES OF MA 12

5.1 Computing the min. of gk 13

5.2 Computing the average of gk 14

CHAPTER 6 OPTIMIZATIONS 15

6.1 Learning the alphabet symbols incrementally 15

6.2 Double check the learned min./max. value 16

CHAPTER 7 RUNNING EXAMPLE: CALCULATOR 17

CHAPTER 8 CALCULATOR EXPERIMENT 21

8.1 Calculator 21

8.2 Considerations on the choice of distribution 22

8.3 Incremental alphabet refinement 23

8.4 Distribution of the execution time 23

CHAPTER 9 OPERATING SYSTEM SCHEDULING EXPERIMENT 25

CHAPTER 10 MISSIONARIES AND CANNIBALS EXPERIMENT 26

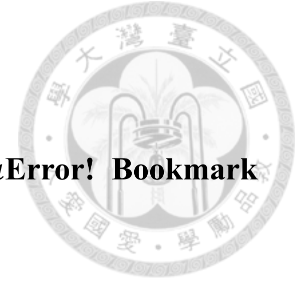
doi:10.6342/NTU201701657

CHAPTER 11 AMOUNT OF DATA TRANSMISSION IN A WEBSITE 28

Summary.. 29

REFERENCE 30

doi:10.6342/NTU201701657



LIST OF FIGURES

Figure 1 An MA computes the number of occurrences of symbol a**Error! Bookmark not defined.**

Figure 2 The learning algorithm for MA 8

Figure 3 Replacing equivalence query with sampling. It assumes the following additional input: a distribution D over $\Sigma *$, the parameters $0 < \varepsilon, \delta < 1.$ 9

Figure 4 Overview 10

Figure 5 The PAC learning algorithm for MA that does not require to know the alphabet beforehand. It assumes the following additional input: a distribution D over words constructed from an unknown finite alphabet, the parameters $0 < \varepsilon, \delta < 1$ and the initial value $\Sigma = \emptyset.$ 15

Figure 6 Iteration 1 18

Figure 7 Iteration2 19

doi:10.6342/NTU201701657

LIST OF TABLES

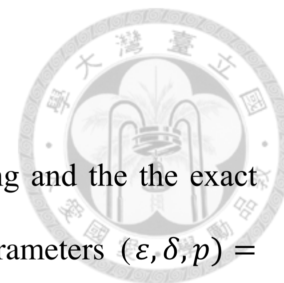


Table 1 Comparing the approximate average computed via learning and the the exact answer obtained directly from the calculator. The parameters $(\varepsilon, \delta, p) = (0.1, 0.9, 0.2)$ 20

Table 2 Comparing the performance of natural and random alphabet mappings. The parameters $(\varepsilon, \delta, p) = (0.2, 0.8, 0.1)$. 21

Table 3 Comparing the performance using different sampling distributions. $(\varepsilon, \delta) = (0.2, 0.8)$. We use the random alphabet mapping. 23

Table 4 The time used in different steps of MA learning. 24

Table 5 Performance on "Operating System Scheduling". The parameters $(\varepsilon, \delta, p) = (0.1, 0.9, 0.2)$. The row "Enumeration" is obtained by enumerating all words of length k. 25

Table 6 Performance on "Missionaries and Cannibals". The parameters $(\varepsilon, \delta, p) =$ $0.1, 0.9, 0.2$. The row "Enumeration" is obtained by enumerating all words of length k. 27

Table 7 Performance on the "Amount of Data Transmission in a Website" problem. The parameters $(\varepsilon, \delta, p) =$ $0.1, 0.9, 0.2$. Here the numbers are in byte and the size of alphabet of the learned MA is 16. 28

doi:10.6342/NTU201701657

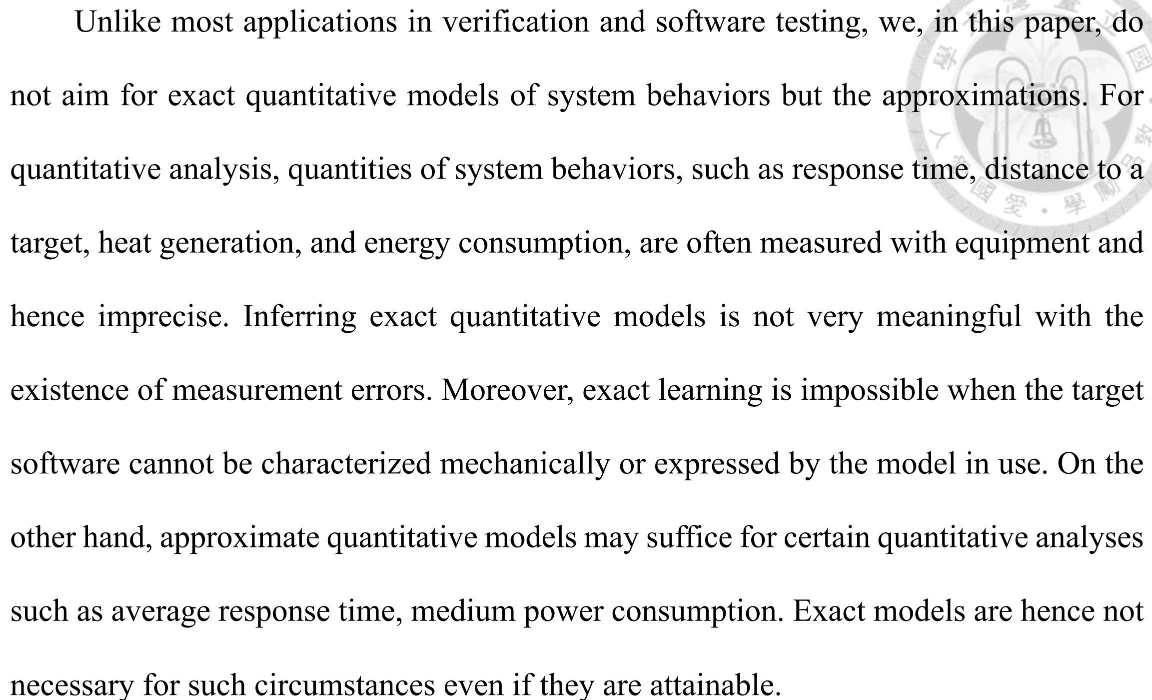
Chapter 1 Introduction

1.1 Motivation

When software developed, limited resources such as time and space are crucial in the stage od system evaluation. Quantitative analysis of system behaviors gradually has become a consequential research issue recently [12,13]. Normally, models are required to describe system behaviors when conducting analysis. Quantitative model construction, however, can be as hard as the analysis itself. Consider analyzing the amount of data transmission in web browsing. When users click on hyperlinks, new pages are generated and sent from servers through Internet. Suppose we want to estimate the average amount of data transmission from one website over k hyperlink clicks. In most cases, the first thing to do is to construct a system model with a suitable abstraction. However, it is not immediate clear how to construct an abstract quantitative model for such system behaviors systematically and, even automatically.

We apply a probably approximately correct (PAC) learning algorithm which is design for multiplicity automata to generate abstract models for quantitative analysis of system behaviors. Multiplicity automata are the class of weighted automata over the semi-ring $(\mathbb{R}, +, \times, 0, 1)$. We also model system behaviors by words in the alphabet of a multiplicity automaton. The quantity, such as the amount of data transmission, associated with the system behavior is hence the result of the automaton on the input word. Through a teacher who simulates the system on certain input and measures the quantity of interest, the learning algorithm cab generate a multiplicity automaton as an approximation to quantitative system behaviors with a statistical guarantee.

doi:10.6342/NTU201701657



Unlike most applications in verification and software testing, we, in this paper, do not aim for exact quantitative models of system behaviors but the approximations. For quantitative analysis, quantities of system behaviors, such as response time, distance to a target, heat generation, and energy consumption, are often measured with equipment and hence imprecise. Inferring exact quantitative models is not very meaningful with the existence of measurement errors. Moreover, exact learning is impossible when the target software cannot be characterized mechanically or expressed by the model in use. On the other hand, approximate quantitative models may suffice for certain quantitative analyses such as average response time, medium power consumption. Exact models are hence not necessary for such circumstances even if they are attainable.

After a multiplicity automaton is generated, we demonstrate how to analyze the maximum and average values of system behaviors with a fixed length k in Section 5. The transitions of a multiplicity automaton with the alphabet of size n can be represented by a matrix form whose entries are polynomials of degree n. For behaviors of length k, they can also be represented by a multivariate polynomial of degree kn. The polynomial enables us to conduct a variety of quantitative analysis with standard mathematical tools. For example, we can calculate the maximum amount of data transmission for k hyperlink clicks in web browsing by gradient descent, genetic algorithms, simulated annealing or other local maxima searching methods.

Several practical issues on applying learning to quantitative analysis are addressed in Section 6. The previous algorithm [2] assumes a fixed alphabet symbol size. In the real world, the alphabet may not be known in advance. We demonstrate an algorithm which increment the alphabet symbol size when necessary. Besides, note that the generated quantitative model is an approximation, some meaningless values, such as a negative

doi:10.6342/NTU201701657

amount of data transmission, on certain behaviors might exist due to its inaccuracy. In this paper, we give some amendments to prevent those meaningless analysis results.

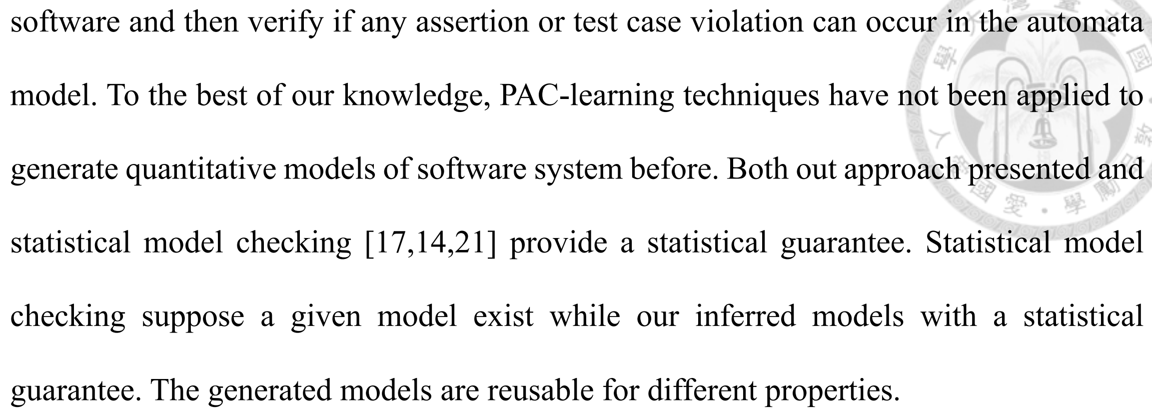
The approach that we presented in this paper has several advantages. For instance, once the mapping between words and system behaviors has been decided, our approach runs fully automatically. After the quantitative models are generated, different analyses can be computed on them without further simulation. For example, suppose we aim to analyze the average amount of data transmission for $1, \ldots, k$ hyperlink clicks from the home page. These k analyses can all be counted by the same inferred approximate model. The quantitative model can also be reused for other types of analyses or applications, such as the maximum amount of data transmission. The experiment results in Section 8 suppose that the estimation obtained by our learning-based approach is almost as accurate as the exact reference answer produced by a brute-force enumeration.

1.2 Related Work

Exact learning algorithms for classical automata was first proposed by Dana Angluin [2]. The result has been generalized to the class of multiplicity automata in [5,6]. Also, the concept of probably approximately correct (PAC) learning was first designed by Valian in [19]. The idea of transform an exact learning algorithm to a probably approximately correct (PAC) learning algorithm was presented in [3].

PAC learning has been applied to software testing and verification [20,11,9]. The work in [20] consider the problem from a theoretical aspect. It focuses on issues related to the lower bound on the number of required queries to generate a system model. The work in [11] tests if the output of a graph-manipulating program is bipartite, k-colorable…etc. The paper of [9] apply PAC-learning to generate a model of a computer

doi:10.6342/NTU201701657



software and then verify if any assertion or test case violation can occur in the automata model. To the best of our knowledge, PAC-learning techniques have not been applied to generate quantitative models of software system before. Both out approach presented and statistical model checking [17,14,21] provide a statistical guarantee. Statistical model checking suppose a given model exist while our inferred models with a statistical guarantee. The generated models are reusable for different properties.

1.3 Contribution

The contributions of our works are as bellowed.

(a) A framework that automatically generates quantitative models from learning with statistical guarantee.

(b) Effective and Efficient algorithm designed to check useful quantitative properties of multiplicity automata.

(c) Down to detail analysis of the capacity of the learning algorithm when applied to quantitative models' construction and suggestion on effective optimizations.

doi:10.6342/NTU201701657

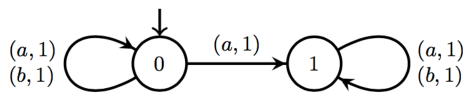
Chapter 2 Preliminaries

In this paper, we assume $\mathbb{N}$ is the set of natural numbers. Matrices and vectors are all over the field of real number $\mathbb{R}$. For a Matrix M, $[M]_{(i,*)}$ is the i-th row of M and $[M]_{(i,j)}$ is the entry at row i and column j. For vector $\boldsymbol{u}$, $\boldsymbol{u}(i)$ is its i-th entry. In this paper, we suppose all vectors are column vectors. We use $\mathbb{R}^{m\times n}$ and $\mathbb{R}^{k}$ to denote the sets of matrices of size $m\times n$ and column vectors of size k, respectively. The product of two matrices M_1, M_2 is denoted as M_1M_2 and the product of k copies of M is denoted as M_k. We use a,b,c,d to denote symbols, w,x,y to denote words, λ to denote the empty word, and v to denote variables. The concatenation of two words x,y is denoted as $x\cdot y$. The set of integers $\{k \mid m \leq k \leq n\}$ is denoted as $[m,n]$ and $[n]$ is a shorthand for $[1,n]$.

2.1 Multiplicity Automata

A multiplicity automaton (MA) $A=(M,b)$ over a finite alphabet $\sum$ is represented as a set of transition matrices $M=\{M_a \in \mathbb{R}^{n\times n} | a \in \sum\}$ (one matrix for each symbol in $\sum$) and an output vector $\boldsymbol{b} \in \mathbb{R}^n$. The output of an MA A cor- responding to a word $\omega = d_1d_2\cdots d_m \in \sum^+$ is $A(\omega) = [M_{d_1}M_{d_2}\cdots M_{d_m}]_{(1,*)}\boldsymbol{b}$ and $A(\lambda) = b(1)$. Intuitively, the entry $[M_a]_{(i,j)}$ the weight of the transition from state qi to qj with symbol a and $\boldsymbol{b}(i)$ is the weight of the state q_i. The initial state is q_1. The output of an MA w.r.t. a word w is the sum of the weight of all runs (sequences of transitions) corresponding to ω, where the weight of a run is the product of the weights of the last state and all transitions in the run. An example of an MA from the view of a set of matrices and also the view of a labeled state-transition system is given in Figure 1.

doi:10.6342/NTU201701657



$M_a = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, $M_b = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, (Transition Matrices)

$\boldsymbol{b} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ (Output Vector)

Figure 1 An MA computes the number of occurrences of symbol a

2.2 Hankel Matrix

The Hankel matrix (HM) of a function $f : \Sigma * \to \mathbb{R}$ is an infinite matrix F indexed with words from $\Sigma *$ such that $[F]_{(x,y)} = f(x \cdot y)$. Let $f : \Sigma * \to \mathbb{R}$ be a function with the corresponding HM F. For short, we use the rank of f, denoted rank (f), to mean the rank of F. We say an MA A is equivalent to f iff $\forall w \in \Sigma *: A(w) = f(w)$. It has been shown in [8,10] that if $r = rank(f)$ is finite then the smallest MA A equivalent to f has r states. More concretely, let $[F]_{(x_1,*)}, [F]_{(x_2,*)}, \ldots, [F]_{(x_r,*)}$ be r independent rows of F with $x_1 = \lambda$. One can construct an equivalent MA $A = (M, b)$ from F as follows. The output vector b is $[F]_{(x_1,\lambda)}, [F]_{(x_2,\lambda)}, \ldots, [F]_{(x_r,\lambda)}$. The transition matrices Ma $\in$ M can be obtained by solving the following equation for each $a \in \Sigma, i \in [r]$:

$$[F]_{(xi \cdot a,*)} = \sum_{j \in [r]} [M_a]_{(i,j)} [F]_{(xj,*)}. \quad - \quad (1)$$

Intuitively, Equation (1) states that the weight from the state represented by the word $x_i \cdot a$ to any state q is equivalent to the sum of the weights from state q_i to state q_j via the symbol a multiplies the weight from qj to q for all $j \in [r]$.

doi:10.6342/NTU201701657

Chapter 3 Learning Algorithm of MA

Now we have all the building blocks required to describe the learning algorithm for MA proposed by Beimel et al. [4], under the minimal adequate teacher (MAT) model by Angluin [2]. The MAT model assumes the existence of a teacher an- swering two types of queries about a function $f : \Sigma * \rightarrow \mathbb{R}$: (a) On membership queries of a word ω, denoted Mem(w), the teacher replies $f(\omega)$. (b) On equivalence queries of an MA A_h, denoted $Equ(A_h)$, the teacher replies true when A is equivalent to f. Otherwise, it replies false accompanying with a word ω s.t. $A_h(\omega) \neq f(\omega)$. Let F be the HM of the target function f. When $r = rank(f)$ is finite, it is sufficient to characterize f using an $r \times r$ sub-matrix of F (with rank r) [4]. The learning algorithm (in Figure 2) tries to find such an $r \times r$ matrix. Assume that the rank of the target function f is finite and let $r = rank(f)$. For the MA learning algorithm in Figure 2, the content $[F_Y]_{(x,*)}$, $[F_Y]_{(x \cdot a,*)}$ can be obtained by $r(r + r|\Sigma|)$ membership queries. The existence of a prefix satisfying conditions (a) and (b) is guaranteed by Claim 3.1 of [4] and it takes only polynomially many membership queries to find such a prefix. Observe that adding $y_l + 1$ to Y is sufficient to make the row of $x_l + 1$ independent with all other rows in X. The learning algorithm will find an MA with r states that is equivalent to f in r iterations.

3.1 PAC Learning

The MA learning algorithm assumes a teacher who can answer equivalence queries. This assumption is invalid in many practical settings. Angluin [3] showed that even if we substitute equivalence testing with sampling, we can still make statistical claims about the difference between the target and inferred model.

doi:10.6342/NTU201701657

Assume the target function for MA learning is $f : \Sigma * \rightarrow \mathbb{R}$ and a probability distribution D over $\Sigma *$ is given. We use $\varphi(\omega)$ to denote that the inferred MA Ah and f are consistent on ω, i.e., $A_h(\omega) = f(\omega)$. The term $Prob_{\omega \Leftarrow D}[\neg\varphi(\omega)]$ denotes the probability that $\varphi(\omega)$ is false for w chosen randomly according to D. For a hypothesis of the form

$$H : Prob_{\omega \Leftarrow D}[\neg\varphi(\omega)] \leq \varepsilon$$

The algorithm that we presented in this paper is as in Figure 2.

Input: The alphabet Σ and a teacher answers $Mem(w)$ and $Equ(\mathcal{A})$ about f.
Init: $x_1 \mapsto \lambda; y_1 \mapsto \lambda; X \mapsto \{x_1\}; Y \mapsto \{y_1\}; l \mapsto 1$.
Step(I): Generate a candidate MA $\mathcal{A}_h$:

- The output vector is $(Mem(x_1), \ldots, Mem(x_l))$.
- Let F_Y be a sub-matrix of F obtained by restricting the columns to the index Y. Observe that each row of F_Y has l elements, but F_Y has infinitely many rows. For each $a \in \Sigma$, the transition matrix M_a can be obtained by solving for each $i \in [l]$ the following equation, which is an adaptation of Equation (1):

$$[F_Y]_{(x_i \cdot a, *)} = \sum_{j \in [l]} [M_a]_{(i,j)} [F_Y]_{(x_j, *)}.$$

Step(II): Ask an equivalence query on $\mathcal{A}_h$:

- If the answer is true, halt and output $\mathcal{A}_h$.
- Otherwise, the teacher returns a word w s.t. $\mathcal{A}_h(w) \neq f(w)$.
- Analyze w and find the pair (x_{l+1}, y_{l+1}) as follows:
  - ∗ Search for a prefix $d_1 \cdots d_k a$ of w satisfying the conditions:
    (a) $[F_Y]_{(d_1 \cdots d_k, *)} = \sum_{j \in [l]} [M_{d_1} \ldots M_{d_k}]_{(1,j)} [F_Y]_{(x_j, *)}$
    (b) $\exists y \in Y : [F_Y]_{(d_1 \cdots d_k a, y)} \neq \sum_{j \in [l]} [M_{d_1} \ldots M_{d_k}]_{(1,j)} [F_Y]_{(x_j \cdot a, y)}$
  - ∗ Assign $x_{l+1} := d_1 \cdots d_k$ and $y_{l+1} := a \cdot y$.
- Reassign $X \mapsto X \cup \{x_{l+1}\}; Y \mapsto Y \cup \{y_{l+1}\}; l \mapsto l + 1$ and goto Step(I).

Figure 2 The learning algorithm for MA

We call ε the error parameter and use confidence to denote the least probability that the hypothesis H is correct. We say that an inferred MA is probably approximately correct (PAC) [19] w.r.t. ε and δ, denoted $PAC(\varepsilon, \delta)$, if H holds with confidence δ. In the example of estimating the amount of data transmission, $f(\omega)$ denotes the actual amount of data transmission with the input ω and A_h is the inferred MA. Consider the uniform

doi:10.6342/NTU201701657

distribution D_k over all words of length k and $(\varepsilon, \delta) = (0.1, 0.9)$. We say A_h is $PAC\ (\varepsilon, \delta)$ if with probability at least 90%, the probability that $f(\omega)$ and $A_h(\omega)$ are different is bounded by 10% when ω is chosen uniformly from words of length k. The task of an equivalence query $Equ(A_h)$ is changed from checking exact equivalence to checking approximate equivalence. More concretely, Step(II) in Figure 2 is replaced with the one in Figure 3.

Step(II): Simulate the equivalence query on $\mathcal{A}_h$ by sampling:
- Let $i = 1; n_l = \left\lceil \frac{1}{\epsilon}\left(\ln \frac{1}{1-\delta} + l \ln 2\right)\right\rceil$.
- Repeat the following steps until $i > n_l$:
  1: Sample a word w according to D and reassign $i := i + 1$.
  2: If $\mathcal{A}_h(w) \neq f(w)$, analyze w and find a pair (x_{l+1}, y_{l+1}) using the approach in Figure 2, reassign $X \mapsto X \cup \{x_{l+1}\}; Y \mapsto Y \cup \{y_{l+1}\}; l \mapsto l + 1$, and goto (I).
- Halt and output $\mathcal{A}_h$.

Figure 3 Replacing equivalence query with sampling. It assumes the following additional input: a distribution D over Σ^*, the parameters $0 < \varepsilon, \delta < 1$.

The teacher answers the i-th equivalence query by picking ni samples accord- ing to D and testing if $A_h(\omega) = f(\omega)$ for all samples ω. The number of samples n_i needed to establish that Ah is $PAC\ (\varepsilon, \delta)$ is given by Angluin in [3]. Note that the target function f is not necessary of a finite rank. When f is of an infinite rank, the learning algorithm can still infer an MA A approximating f with a statistical guarantee.

doi:10.6342/NTU201701657

Chapter 4 Overview

The picture shown in Fig. 4 is the abstraction of the learning algorithm.

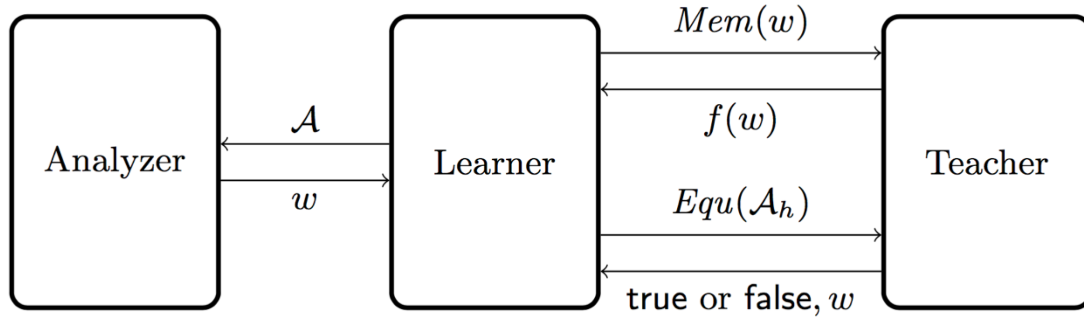
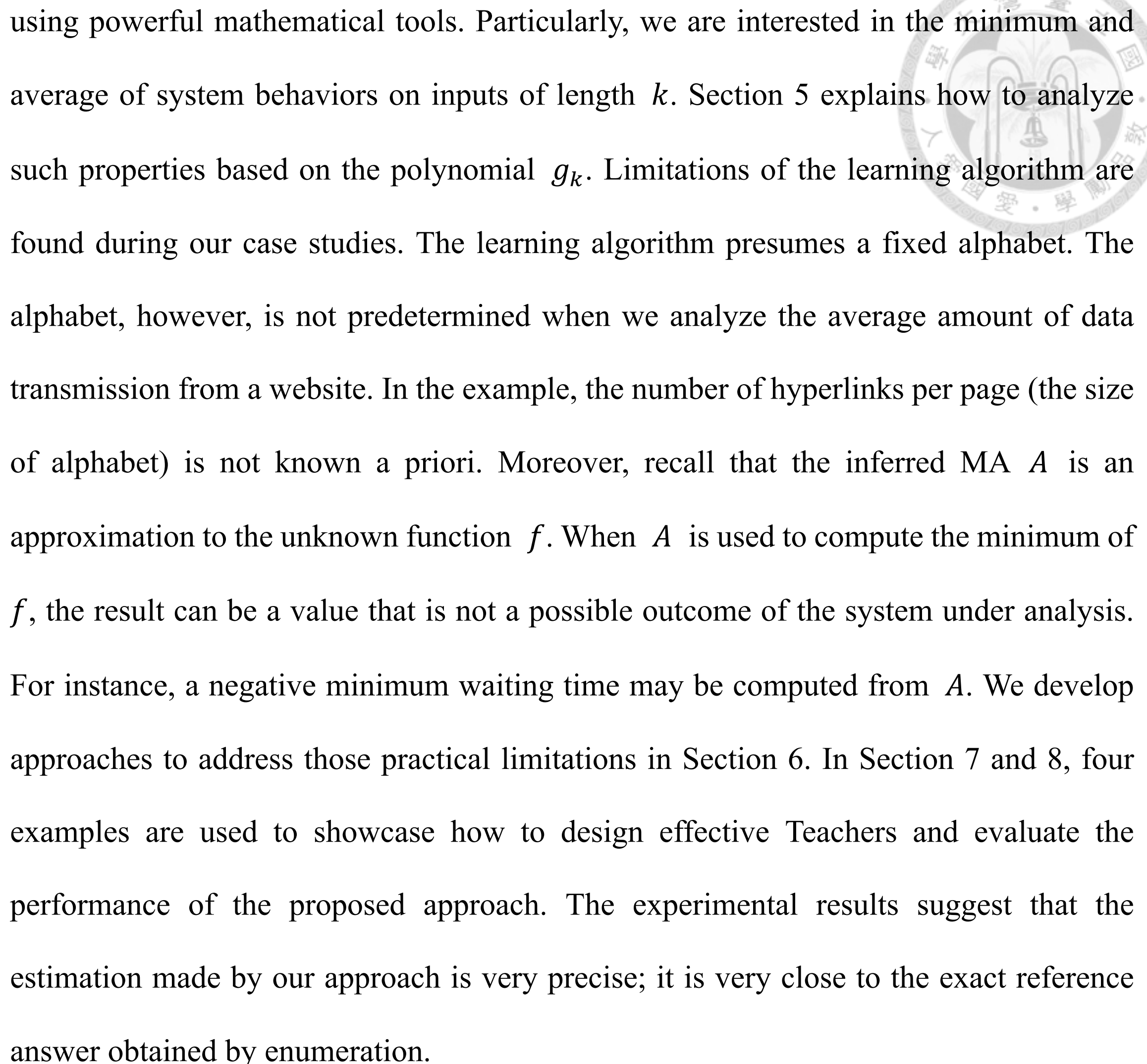


Figure 4 Overview

The learning algorithm for MA will be applied to construct a quantitative model of system behaviors. Fix an alphabet Σ for system actions. Assume that the system behavior on $\omega \in \Sigma*$ can be modeled by the quantity $f(\omega) \in \mathbb{R}$ for some unknown $f : \Sigma* \to \mathbb{R}$. Figure 4 gives an overview of our framework. In the figure, the Teacher measures the quantity $f(\omega)$ by simulating the system on w. On a membership query $Mem(\omega)$, the Teacher answers the query by measuring the quantity $f(\omega)$. On an equivalence query $Equ(A_h)$, the Teacher checks if $A_h(\omega)$ and the measured quantity $f(\omega)$ coincide on a number of randomly chosen ω. If so, the Teacher concludes that the MA A_h represents the unknown function f with a statistical guarantee and the Learner will pass A_h to the Analyzer for further analysis. Otherwise, the Teacher returns w_0 with $A_h(w_0) \neq f(w_0)$. Once an approximation A to the unknown function f is obtained from the Learner, the Analyzer transforms A to a multivariant polynomial $g_k(d_1 d_2 \cdots d_k \in \Sigma^k)$ which computes $A(d_1 d_2 \cdots d_k)$ for any $d_1 d_2 \cdots d_k \in \Sigma^k$. The transformation to the polynomial form allows us to perform various quantitative analyses

doi:10.6342/NTU201701657



using powerful mathematical tools. Particularly, we are interested in the minimum and average of system behaviors on inputs of length k. Section 5 explains how to analyze such properties based on the polynomial g_k. Limitations of the learning algorithm are found during our case studies. The learning algorithm presumes a fixed alphabet. The alphabet, however, is not predetermined when we analyze the average amount of data transmission from a website. In the example, the number of hyperlinks per page (the size of alphabet) is not known a priori. Moreover, recall that the inferred MA A is an approximation to the unknown function f. When A is used to compute the minimum of f, the result can be a value that is not a possible outcome of the system under analysis. For instance, a negative minimum waiting time may be computed from A. We develop approaches to address those practical limitations in Section 6. In Section 7 and 8, four examples are used to showcase how to design effective Teachers and evaluate the performance of the proposed approach. The experimental results suggest that the estimation made by our approach is very precise; it is very close to the exact reference answer obtained by enumeration.

doi:10.6342/NTU201701657

Chapter 5 Analyzing Properties of MA

When the learning algorithm finds an MA A for the target system, the next step is to analyze the quantitative properties of A. Two interesting quantitative properties of MA are identified: (1) the minimum output value of an MA from an input of length k and (2) the average output value of an MA from all inputs of length k. A naive way to compute the minimum or average output values of a given MA is to enumerate all inputs of length k and compute the corresponding output. It is easy to see that the naive approach cannot scale to a large k. So our goal is to develop more efficient algorithms to compute these values. Assuming that the Analyzer receives an MA $A = (M, b)$, where $M = \{M_a \in \mathbb{R}^{n\times n} \mid a \in \Sigma\}$ and $\Sigma \subset \mathbb{N}$,from the Learner. It will transform A to a multivariate polynomial $g_k(d_1 d_2 \cdots d_k)\colon \mathbb{R}^{\mathrm{k}} \to \mathbb{R}$ that outputs the value of $A(d_1 d_2 \cdots d_k)$ when $d_1 d_2 \cdots d_k \in \Sigma^k$.

The transformation is similar to the one in [4] using interpolation. It first computes $p(v)$, an $n \times n$ matrix of polynomials over the variable v, as follows.

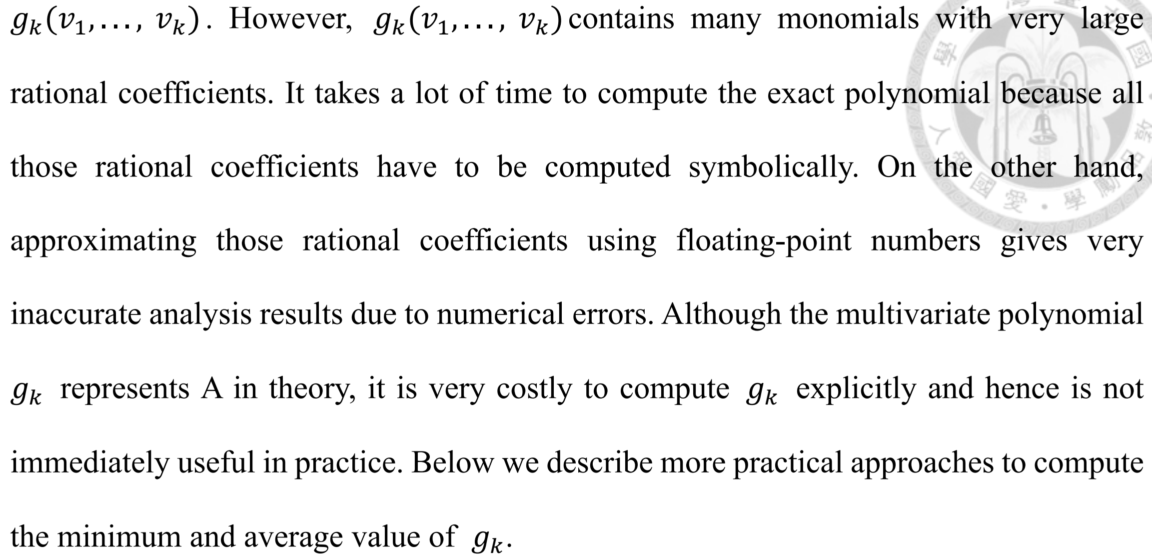
$$p(v) \ = \ \sum_{a\in\Sigma}(\prod_{b\in\Sigma\backslash\{a\}} \frac{v-b}{a-b} M_a)$$

Example 1. Consider the MA in Figure 1. We use 0, 1 to represent a, b, respectively. Then $p(v) = \left(\frac{v-1}{0-1} M_a\right) + \left(\frac{v-0}{1-0} M_b\right) = \begin{bmatrix} 1 & (1-v) \\ 0 & 1 \end{bmatrix}$.

Observe that $\forall a \in \Sigma : p(a) = M_a$. Then $g_k(v_1, \ldots, v_k)$ is defined as $g_k(v_1, \ldots, v_k) \ = \ [p(v_1)p(v_2)\ldots p(v_k)]_{(1,*)}\boldsymbol{b}$

It is easy to see that $g_k(v_1, \ldots,\ v_k)$ is indeed a multivariate polynomial satisfying all requirements specified above. In principle, standard calculus techniques can be applied to analyze properties (such as optimal values or average) of the multivariate polynomial

doi:10.6342/NTU201701657



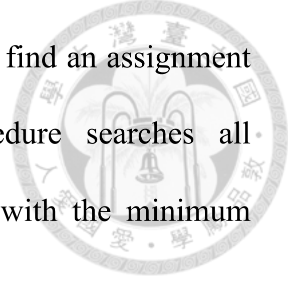
$g_k(v_1, \dots, v_k)$. However, $g_k(v_1, \dots, v_k)$ contains many monomials with very large rational coefficients. It takes a lot of time to compute the exact polynomial because all those rational coefficients have to be computed symbolically. On the other hand, approximating those rational coefficients using floating-point numbers gives very inaccurate analysis results due to numerical errors. Although the multivariate polynomial g_k represents A in theory, it is very costly to compute g_k explicitly and hence is not immediately useful in practice. Below we describe more practical approaches to compute the minimum and average value of g_k.

5.1 Computing the min. of $\mathbf{g_k}$

The global optimization problem of multivariate polynomial is known to be very difficult. It is already NP-hard when the degree is 4 [15]. Here we suggest to use the gradient descent (GD) algorithm or any similar algorithm1 to find a local minimum of g_k instead. Let $V = \{v_i \mid i \in [k]\}$. Intuitively, the GD algorithm begins with an arbitrarily chosen initial assignment $\eta : V \to \mathbb{R}$. It searches in g_k the direction from η leading to the steepest downward gradient and picks another assignment by moving from η toward the chosen direction for a distance. The steeper the gradient is, the longer the distance is. The algorithm repeats the above procedure to obtain better assignments. It terminates when, e.g., the distance to move becomes very small, which indicates that an assignment close to a local minimum is reached. Note that the GD algorithm does not need the polynomial g_k explicitly. It only requires the values of g_k on the selected assignments. Since $g_k(d_1, \dots, d_k) = [p(d_1)p(d_2) \dots p(d_k)]_{(1,*)}\boldsymbol{b}$, we use the MA A to compute the values of $p(d_1), p(d_2), \dots, p(d_k)$ on given assignments. When the GD algorithm is applied to our analysis, it begins with an arbitrarily chosen assignment from

doi:10.6342/NTU201701657

V to the discrete domain Σ^k. However, the GD algorithm may still find an assignment η' outside Σ^k when it terminates. In this case, our procedure searches all "neighboring" assignments to η' over Σ^k and pick the one with the minimum output w.r.t g^k.



5.2 Computing the average of g_k

The average value can be obtained by computing the sum using the following formula and then dividing it by $|\Sigma|^k$.

$$\sum_{d_1\in\Sigma}\sum_{d_2\in\Sigma}\cdots\sum_{d_k\in\Sigma} g_k(d_1,\ldots,d_k)$$

$$=\sum_{d_1\in\Sigma}\sum_{d_2\in\Sigma}\cdots\sum_{d_k\in\Sigma}[p(d_1)p(d_2)\ldots p(d_k)]_{(1,*)}b-(2)$$

$$=\left[\left(\sum_{d_1\in\Sigma}p(d_1)\right)\left(\sum_{d_2\in\Sigma}p(d_2)\right)\cdots\left(\sum_{d_k\in\Sigma}p(d_k)\right)\right]_{(1,*)}\boldsymbol{b}$$

$$=\left[\left(\sum_{d_1\in\Sigma}M_d\right)^k\right]_{(1,*)}\boldsymbol{b}-(3)$$

In our implementation, we use a similar gradient-based algorithm, called *sequential quadratic programming (SQP)* [7], implemented in the *fmincon* function of Matlab.

Sometimes we are only interested in the average value w.r.t. a subset S of Σ. Such an average value can be computed by replacing Σ in (3) with S. Observe that the computation of (2) is more expensive than (3). The former uses $k|\Sigma|^k$ matrix product operations, while the latter uses only k product operations.

doi:10.6342/NTU201701657

Chapter 6 Optimizations

In this section, approaches to address some practical limitations of our learning-based algorithm are discussed.

6.1 Learning the alphabet symbols incrementally

Recall that the MA learning algorithm assumes a finite alphabet Σ. The assumption does not hold for systems such as a website. We propose an adaption to the learning algorithm to eliminate the assumption. The main idea is to incrementally build the alphabet Σ. Initially, we assume $\Sigma = \emptyset$. The algorithm also works when Σ is a nonempty subset of system actions. If a word ω sampled according to D contains new symbols, i.e., $sym(w)\ not \subseteq \Sigma$, we reassign $\Sigma := \Sigma \cup sym(\omega)$ and use **Step(I)** of the learning algorithm to rebuild Ah. The update of the alphabet Σ will eventually terminate, provided that the distribution D is over words constructed from a finite alphabet. The algorithm is obtained by modifying the **Step(II)** in Figure 2 to the one in Figure 5. Later we will see in Section 8.1 that applying the optimization improves the overall performance by roughly 10% even for systems where Σ can be predetermined.

Step(II): Simulate the equivalence query on $\mathcal{A}_h$ by sampling:

- Let $i = 1; n_l = \left\lceil \frac{1}{\epsilon}\left(\ln \frac{1}{1-\delta} + l \ln 2\right)\right\rceil$.
- Repeat the following steps until $i > n_l$:
  1: Sample a word w according to D and reassign $i := i + 1$.
  2: If $sym(w) \not\subseteq \Sigma$, reassign $\Sigma := \Sigma \cup sym(w)$ and rebuild $\mathcal{A}_h$ using the procedure in (I). Here $sym(w)$ denotes the set of symbols in w.
  3: If $\mathcal{A}_h(w) \neq f(w)$, analyze w and find a pair (x_{l+1}, y_{l+1}) using the approach in Figure 2, reassign $X \mapsto X \cup \{x_{l+1}\}; Y \mapsto Y \cup \{y_{l+1}\}; l \mapsto l + 1$, and goto (I).
- Halt and output $\mathcal{A}_h$.

Figure 5 The PAC learning algorithm for MA that does not require to know the alphabet beforehand. It assumes the following additional input: a distribution D over words

doi:10.6342/NTU201701657

constructed from an unknown finite alphabet, the parameters $0 < \varepsilon, \delta < 1$ and the initial value $\Sigma = \emptyset$.

6.2 Double check the learned min./max. value

Let A be the MA inferred by the learning algorithm and $f : \Sigma* \to \mathbb{R}$ be an unknown function representing the behavior of the system under analysis. Assume that our approach finds a minimum value on A with the input $\overline{\omega}$. Since A is an approximation of f, it can be the case that $A(\overline{\omega}) \neq f(\overline{\omega})$. Sometimes, a result of this kind is mean- ingless, e.g., the result can be a negative amount of people. In such a case, we suggest to return $\overline{\omega}$ as a counterexample to the MA learning algorithm to refine the conjecture further. The immediate benefit of the optimization is that we can guarantee that the model and the system are consistent at the inferred minimum/maximum value.

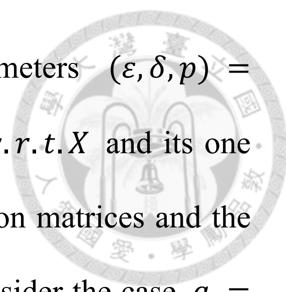
doi:10.6342/NTU201701657

Chapter 7 Running Example: Calculator

In this section, we demonstrate how our approach works using a simple but concrete example, a "calculator" with numeral buttons 0 to 9 and operator buttons + and −. We want to compute the average and maximal output values the calculator can produce with an input of length k. Here a natural choice is to map each button to a symbol in [0, 11] because we have 12 buttons in total. We use the mapping that buttons 0 to 9 are mapped to the corresponding number in [0,9] and +,− are mapped to 10,11, respectively. We use an underline to emphasize the segmentation of symbols, e.g., to distinguish 1 0 and 10. A word in $[0,11]*$ is evaluated in the same way as Matlab does. For example, $[\![\underline{3}\ \underline{3}\ \underline{4}]\!] = 334$, $[\![\underline{3}\ \underline{10}\ \underline{4}]\!]$ =7 (interpreted as 3+7), and $[\![\underline{1}\ \underline{10}\ \underline{11}\ \underline{2}]\!]$ (interpreted as 1+(−2)). Here $[\![\omega]\!]$ is the evaluation of ω in Matlab. For an incomplete expression (e.g., the empty word λ or $\underline{10}\ \underline{11}$, which is interpreted as +−), its evaluation is 1.

The PAC-learning algorithm for MA requires a distribution over words in $[0,11]*$ that will be used for sampling. We use the so-called monkey distribution with a stop probability p. The distribution tries to simulate the behavior of a monkey playing a system. The monkey has no preference on which button to push and hence each symbol is assumed to have the same chance to be pushed. There is a probability p (checked after each button pushing) that the monkey is bored and decides to stop pushing more buttons. A similar idea has been used in software testing under the name "monkey testing", which is included as a stan- dard testing tool in Android Studio [1]. The monkey distribution can be viewed as a generalization of the geometric distribution in probability theory to finite words. The average length of word sampled by the monkey distribution is $1/p$. We demonstrate the first two iterations of applying the MA learning algorithm to learning

doi:10.6342/NTU201701657



the calculator model in Figure 6 and 7. Assume the parameters $(\varepsilon, \delta, p) = (0.05, 0.9, 0.2)$. On the left of Figure 6, we show the rows of $F_Y\ w.r.t. X$ and its one step extension. These numbers are sufficient to establish all transition matrices and the output vector. For example, now we have $x_1 = \lambda,\ l = 1$ and consider the case $a = 9$ and $i = 1$, from the equation

$$[9] = [F_Y]_{(\lambda \cdot \underline{9}, *)} = \sum_{j \in [l]} [M_a]_{(i,j)} [F_Y]_{(x_j, *)} =$$

$$\sum_{j \in [1]} [M_{\underline{9}}]_{(1,j)} [F_Y]_{(x_j, *)} = [M_{\underline{9}}]_{(1,1)} [F_Y]_{(\lambda, *)} = [M_{\underline{9}}]_{(1,1)} [1],$$

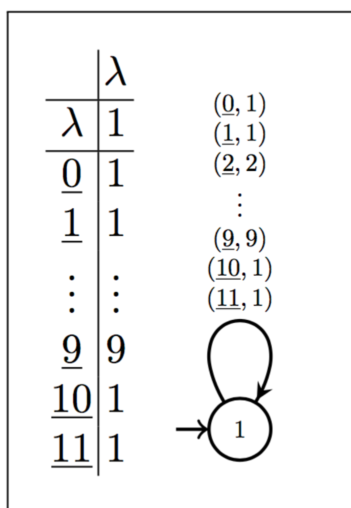
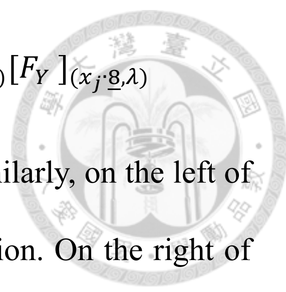


Figure 6 Iteration 1

we can derive $M_9 = [9]$. On the right of Figure 6, we show the first conjectured MA A_{h_1} . The teacher returns the first counterexample $ce_1 = \underline{6}\ \underline{8}$. Observe that $A_{h_1}(ce_1) = 48$ while $f(ce_1) = 68$. By analyzing ce_1, we found its prefix $\underline{6}\ \underline{8}$ satisfies both conditions stated in the step(II) of Figure 2 with $y = \lambda$ as follows.

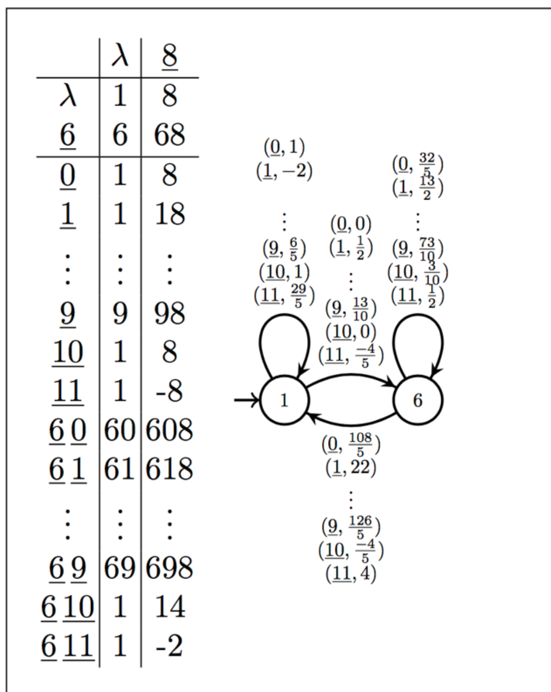
(a) $[F_Y]_{(\underline{6}, *)} = [6] = [M_{\underline{6}}]_{(1,1)} [F_Y]_{(\lambda, *)} = \sum_{j \in [1]} [M_{\underline{6}}]_{(1,1)} [F_Y]_{(\lambda, *)}$

doi:10.6342/NTU201701657



(b) $[F_Y]_{(\underline{6}\,\underline{8},\lambda)} = 68 \neq 48 = [M_{\underline{6}}]_{(1,1)}[F_Y]_{(\lambda\cdot\underline{8},\lambda)} = \sum_{j\in[1]}[M_{\underline{6}}]_{(1,j)}[F_Y]_{(x_j\cdot\underline{8},\lambda)}$

Hence we add $(\underline{6},\underline{8})$ to (X,Y) and proceed to iteration 2. Similarly, on the left of Figure 7, we show the rows of F_Y w.r.t. X and its one step extension. On the right of Figure 7 we show the conjectured MA A_{h_2} . Due to space limit, we leave the details of the construction of A_{h_2} in Appendix A.1. Still, A_{h_2} is incorrect evidenced by the counterexample $ce_2 = \underline{11}\;\underline{11}$. Observe that $A_{h_2}(ce_2) = \frac{127}{5}$ while $f(ce_2) = 1.5$. The learning algorithm will analyze ce_2 and extend the sets X and Y. It repeats the above procedure until finds an MA A_h that is $PAC(\varepsilon,\delta)$. That is, with confidence 90%, if a word w is sampled using the monkey distri- bution, the probability that $A_h(\omega) \neq f(\omega)$ is less than 5%.

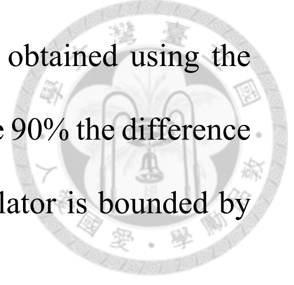


	λ	8
λ	1	8
6	6	68
0	1	8
1	1	18
⋮	⋮	⋮
9	9	98
10	1	8
11	1	-8
6 0	60	608
6 1	61	618
⋮	⋮	⋮
6 9	69	698
6 10	1	14
6 11	1	-2

Figure 7 Iteration2

Notice that although the inferred MA A has a statistical guarantee on the difference to the behaviors of the calculator. The guarantee does not directly carry over to

doi:10.6342/NTU201701657



quantitative properties of MA (such as the minimum and average) obtained using the approaches introduced in Section 5. We can show that with confidence 90% the difference in the average of words of length 4 between A and the actual calculator is bounded by 164.5% (check Appendix A.2 for details).

However, the bound is very loose and hence not that useful. Our experimental results suggest that the quality of MA learned by the algorithm and the inferred quantitative properties are much better than the worst case theoretical bound. In Table 1, we compare the approximate average values obtained via learning and the exact values obtained directly from the calculator.

	Length 5	Length 10	Length 15	Length 20	Length 25	Length 30
Learning	4.8×10^5	4.6×10^{10}	4.4×10^{15}	4.2×10^{20}	4.1×10^{25}	3.8×10^{30}
Exact	4.5×10^5	4.5×10^{10}	4.5×10^{15}	4.5×10^{20}	4.5×10^{25}	4.5×10^{30}
Difference(%)	6%	2%	2%	7%	9%	16%

Table 1 Comparing the approximate average computed via learning and the the exact answer obtained directly from the calculator. The parameters $(\varepsilon, \delta, p) = (0.1, 0.9, 0.2)$.

Recall that the average sample length is $1/p = 5$. We compare the inferred approximation with exact value on length up to 30. To make the result easier to verify, we compute the average of words over the alphabet $[0,9] \subset [0,11]$, Recall that the algorithm in Section 5 allows us to focus on a subset of [0, 11]. All words in $[0,9]^k$ are completed by Matlab expressions and we can easily compute by hand that the average of a length k word in $[0,9]^k$ is 4.5×10^k . Still, we want to emphasize that the learning algorithm and also the inferred MA A is over the complete alphabet [0, 11].

doi:10.6342/NTU201701657

Chapter 8 Calculator Experiment

We first use the calculator example to perform an in-depth evaluation of our approach from different aspects. For instance, we study the performance impact when the incremental alphabet refinement optimization is turned on and off. We then examine the generality of the proposed approach using three more examples: “operating system scheduling”, “missionaries and cannibals”, and “amount of data transmission in a website”. Our implementation is in Matlab and Perl.0

8.1 Calculator

Considerations on the choice of alphabet symbols. Observe that our mapping from buttons to alphabet symbols keeps the natural order of the numeral but- tons. Below we evaluate whether such a mapping is helpful to the performance of our approach. We call the mapping we introduced before the natural map- ping. Here we define the random mapping which assign randomly each button to a number in [0,11]. The result of the experiment is in Table 2, which is the summary of 20 runs for each alphabet mapping. The results in the row “Enumeration” is obtained by a brute-force enumeration of all words w of length 5 and then computing the average of their evaluation in Matlab. The column “#Mem. Queries” is partitioned into three parts: “HM”,“PAC”, and “CE”, which denotes those for filling the HM, PAC-based sampling, and counterexample analysis, respectively. However, based on our observation more than a half of the membership queries are being used for filling the HM which is necessary for the learning process.

doi:10.6342/NTU201701657

	Time in Learning (Sec.)	#Equ. Queries	#Mem. Queries			Length 5	
			HM	PAC	CE	Average	Maximum
Natural	60.01	11	1622	106	615	20152.64	98646.75
Random	122.65	14	2922	168	1038	18460.47	99449.60
Enumeration						20969.34	99999

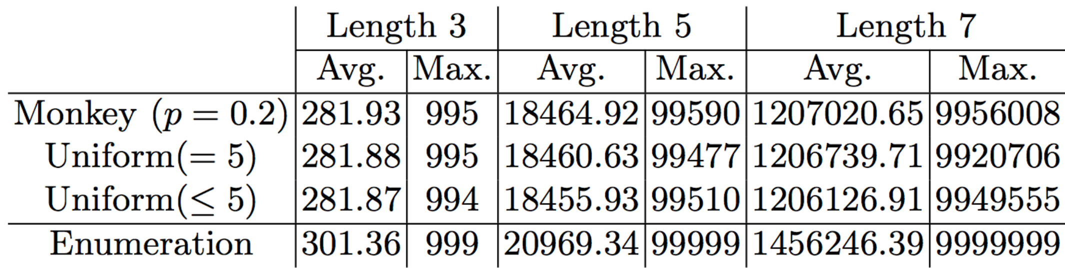
Table 2 Comparing the performance of natural and random alphabet mappings. The parameters $(\varepsilon, \delta, p) = (0.2, 0.8, 0.1)$.

The analysis using the natural mapping is clearly more efficient than the one with random mapping. We believe the reason is that it is easier for the learning algorithm to find "regularity" when the mapping is natural. This is supported by the fact that the size of the MA learned with the natural mapping is significantly smaller than the one with a random mapping. The lesson learned here is that to use a natural mapping when it is possible for the system under analysis.

8.2 Considerations on the choice of distribution

We evaluate the impact of choosing different distributions. Beside the monkey distribution, we introduce the other two distributions. (a) A uniform distribution over all words of length 5. (b) A uniform distribution over all words of length smaller than or equals to 5. The average result of 20 runs of each distribution is in Table 3. We found that our quantitative analysis is very stable w.r.t. the choice of sampling distributions. Observe that a word of length longer than 5 will never be sampled using the two uniform distributions, but the estimated values on length 7 are still very precise. We believe this is due to the fact the MA learning algorithm is very good in generalizing the collected samples. Note that the HM may still contain entries corresponding to words of length longer than 5.

doi:10.6342/NTU201701657



	Length 3		Length 5		Length 7	
	Avg.	Max.	Avg.	Max.	Avg.	Max.
Monkey ($p = 0.2$)	281.93	995	18464.92	99590	1207020.65	9956008
Uniform($= 5$)	281.88	995	18460.63	99477	1206739.71	9920706
Uniform(≤ 5)	281.87	994	18455.93	99510	1206126.91	9949555
Enumeration	301.36	999	20969.34	99999	1456246.39	9999999

Table 3 Comparing the performance using different sampling distributions. $(\varepsilon, \delta) = (0.2, 0.8)$. We use the random alphabet mapping.

8.3 Incremental alphabet refinement

We use a 10 min timeout period and the parameters $(\varepsilon, \delta, p) = (0.2,0.8,0.1)$ to evaluate the performance difference of our approach when the incremental alphabet refinement optimization is turned on and off. We execute 100 MA learning tasks for each setting. If a task cannot be completed within the timeout period, we use 600 sec as its execution time. The setting when the optimization is turned off has 20 timeouts and the average execution time is 166.96 sec. The one with the optimization turned on has only 17 timeouts and the average execution time is 150.99 sec. Here we can see that the gain in execution time with the optimization is roughly 10%.

8.4 Distribution of the execution time

We investigate the performance bottleneck of MA learning. The top 4 time-consuming component are (1) filling the Hankel matrices, (2) building the transition matrices, (3) processing PAC-based equiv- alence queries by sampling, and (4) counterexample analysis. The results are presented in Table 4. We set the error rate to almost zero so the learning al- gorithm will never terminate. The stop probability p is set to 0.1. Beside the standard 12-button calculator, we also tried calculator with 22 and 42

doi:10.6342/NTU201701657

buttons, i.e., with numeral button 0-19 and 0-39, respectively. We list the time spent in iterations 10, 20 , 40, and 80. The result indicates that most of the time is spent in (1) and hence should have the highest priority for further optimizations. For the 80-th iteration of the case $|\Sigma| = 12$, the time spent in PAC-equivalence query dominates the total execution time. The reason is that the inferred MA is already very close to the actual behavior of the calculator. So the teacher needs to sample and test a large number of words before preceding to the next iteration. Also observe that if the time budget is one hour, the learning algorithm can find an MA with more than 40 states even if the alphabet size is 42.

$\|\Sigma\|$	Iter.	Time in seconds				
		Filling the HM	Building the TM	PAC Equ. Query	CE analysis	Total
12	10	11.08	2.91	2.52	11.03	27.60
	20	59.07	15.76	40.74	72.90	188.83
	40	315.00	114.06	241.49	112.90	784.20
	80	1950.94	941.27	2223.31	372.71	5490.04
22	10	19.98	4.91	1.77	8.65	35.36
	20	108.63	28.87	7.10	45.78	160.47
	40	675.50	203.77	38.81	38.67	956.90
	80	5283.39	1725.94	398.66	203.06	7611.33
42	10	46.68	9.19	1.08	5.36	62.33
	20	297.34	57.74	4.79	11.18	371.10
	40	2124.13	412.61	34.18	30.23	2601.25
	80	23672.27	3800.65	263.59	155.37	27892.06

Table 4 The time used in different steps of MA learning.

doi:10.6342/NTU201701657

Chapter 9 Operating System Scheduling Experiment

An operating system (OS) on a uniprocessor machine maintains a queue of processes that are ready to run. Depending on the scheduling policy, the OS may deactivate the running process, insert it into the queue, and then remove some process p from the queue and activate p for a certain time period. In this example, we assume the first come first serve (FCFS) scheduling policy [18].

We are interested in the waiting time of a process (the total time period in which the process is ready to run but not activated). We assume the maximal execution time is 10 time units for all processes and define a set of alphabet $\Sigma = [0, 10]$. Basically, for a word $\omega = a_0 a_1 \dots a_l$, the symbol a_i indicates that at the $i-th$ time unit (1) a new process with execution time a_i is arrived and ready to run if $a_i > 0$, or (2) no new process arrived if $a_i = 0$. The output $f(\omega)$ is computed by simulating the OS under the FCFS policy. Table 5 summaries the results of running the analysis 3 times. The result is surprisingly promising; our analysis is as precise as the result obtained by enumerating all words of length k.

	Time (Sec.)	#Equ. Queries	Length 3		Length 5		Length 7	
			Avg.	Max.	Avg.	Max.	Avg.	Max.
Learning	212.08	22.33	4.50	9.00	9.00	18.00	13.50	27.00
Enumeration			4.50	9.00	9.00	18.00	13.50	27.00

Table 5 Performance on “Operating System Scheduling”. The parameters $(\varepsilon, \delta, p) = (0.1, 0.9, 0.2)$. The row “Enumeration” is obtained by enumerating all words of length k.

doi:10.6342/NTU201701657

Chapter 10 Missionaries and Cannibals Experiment

The missionaries and cannibals example is one of the classical river-crossing problems. In our setting, 3 missionaries and 3 cannibals want to cross a river using a boat under the following constraints: (1) the boat can carry at most 3 people; (2) at least one people is required to row the boat; (3) if there are more cannibals than missionaries present on a bank (or on a boat), then the cannibals will devour the missionaries. To analyze the missionaries and cannibals example with MA, we define a set of alphabet $\Sigma = \{(i,j) \mid i+j \in [3]\}$. For a word $w = (i_0,j_0)(i_1, j_1)\dots(i_l, j_l)$, the symbol (i_k, j_k) indicated that i_k missionaries and j_k cannibals row the boat (1) from the source bank to the destination bank if k is even, or (2) in the other direction if k is odd. Our goal is to estimate the number of people on the destination bank at the $k-th$ step. We encode the number of people on the destination bank in the power of 2. That is, 2^n denotes there are n people on the destination bank. Then moving one person to the destination bank becomes $\times 2$ and removing one becomes $\div 2$.

Observe that by encoding the number of missionaries and cannibals at each bank and the position of the boat as the states of an MA, one can obtain a deterministic MA that precisely computes the number of people on the destination bank. Each alphabet symbol will move the MA from one state to only one target state and update the number of people on the destination bank accordingly using $\times 2$ and $\div 2$. The number of states are bounded by $3^6 \times 2$. The boat has two positions and each person has at most three statuses: at the source, at the destination, and being devoured. So we know that rank of the target function is finite, although its value can be high.

Table 6 summaries the results of running the analysis 20 times. The output $f(\omega)$ is

doi:10.6342/NTU201701657

computed by simulating the move of the boat according to ω. In general, our method produces a very precise estimation on the average output value. In this case, the maximum value we obtained is only sub-optimal. We believe this might due to a special feature of the example: once we made an incorrect step, some missionaries will be devoured and there is no way to resurrect them. So the imprecision of the model will cause a huge impact to the estimated maximum value. Similarly, we obtain the reference answer by enumeration. We only have results up to length 7 because it takes more than 10 hours to compute the reference answer when the length is 7.

	Time (Sec.)	#Equ. Queries	Length 3		Length 5		Length 7	
			Avg.	Max.	Avg.	Max.	Avg.	Max.
Learning	682.09	32	$2^{2.00}$	$2^{4.74}$	$2^{1.99}$	$2^{4.74}$	$2^{1.97}$	$2^{5.05}$
Enumeration			$2^{2.00}$	2^{5}	$2^{1.99}$	2^{6}	$2^{1.97}$	2^{6}

Table 6 Performance on "Missionaries and Cannibals". The parameters $(\varepsilon, \delta, p) = (0.1, 0.9, 0.2)$. The row "Enumeration" is obtained by enumerating all words of length k.

doi:10.6342/NTU201701657

Chapter 11 Amount of Data Transmission in a Website

Average and worst-case response time are important measures of the performance of a website. For static web pages, the response time is usually proportional to the size of the page being transmitted. In the experiment, we estimate the average and maximum size of data transmitted during k-page visits.

Define an initial set of alphabet $\Sigma = [2]$. Basically, a symbol $i \in \Sigma$ indicates the $i-th$ hyperlink in the current web page. A word $\underline{3}\,\underline{4}\,\underline{2}$ denotes the sequence of actions: click the 3rd hyperlink in the first web page, the 4th hyperlink in the next web page, and then the 2nd link in the last web page. Whenever a web page containing k hyperlinks with $k > 2$ is detected during sampling, the alphabet is extended to $[k]$. We use the personal web-site of our colleague as the target to analyze. The result is presented in Table 7.

	Time	#Equ.	Length 3		Length 5		Length 7	
	(Sec.)	Queries	Avg.	Max.	Avg.	Max.	Avg.	Max.
Learning	371.58	4.00	35365.04	73398.33	53207.33	122083.67	67766.23	140557.67
Enumeration			35365.04	77757.00	-	-	-	-

Table 7 Performance on the "Amount of Data Transmission in a Website" problem. The parameters $(\varepsilon, \delta, p) = (0.1, 0.9, 0.2)$. Here the numbers are in byte and the size of alphabet of the learned MA is 16.

We encountered a number of difficulties working on a realistic problem like this. For example, the web server blocks our connection when we make too many requests within a period of time. So we can only send one request per second to avoid being blocked. Subsequently, a membership on a word of length n requires n requests to the website, which costs at least n seconds. Therefore, here we can only offer the exact reference

doi:10.6342/NTU201701657

answer for the case of length 3. We could not offer the reference answer of other lengths because it would require months of time.

Summary

Our work is the first to apply an MA learning algorithm with a PAC guarantee to the context of quantitative analysis. The encouraging experimental results suggest that our approach has tremendous potential. Although the MA learn- ing algorithm terminates only when the target function is of a finite rank, our approach can be applied even when the rank of the target function is infinite. Observe that beside the example "missionaries and cannibals", we do not know if the the target function is of a finite or an infinite rank. Currently, our tool can infer an MA model with 50 to 100 states within one hour, provided that the size of alphabet is below 50. Our implementation is in Matlab and Perl. We believe its performance can be improved using a more efficient programming language.

doi:10.6342/NTU201701657

REFERENCE

1. Android Studio: the Monkey tester. https://developer.android.com/studio/test/monkey.html (2017) Accessed: 2017-01-08.

2. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2) (1987) 87–106

3. Angluin, D.: Queries and concept learning. Machine Learning 2(4) (1988) 319–342

4. Beimel, A., Bergadano, F., Bshouty, N.H., Kushilevitz, E., Varricchio, S.: Learning functions represented as multiplicity automata. JACM 47(3) (2000) 506–530

5. Bergadano, F., Varricchio, S.: Learning behaviors of automata from multiplicity and equivalence queries. SIAM J. Comput. 25(6) (1996) 1268–1280

6. Berstel, J., Reutenauer, C.: Rational Series and Their Languages. Volume 12 of EATCS Monographs. Springer (1988)

7. Boggs, P.T., Tolle, J.W.: Sequential quadratic programming. Acta Numerica 4 (1995) 1–51

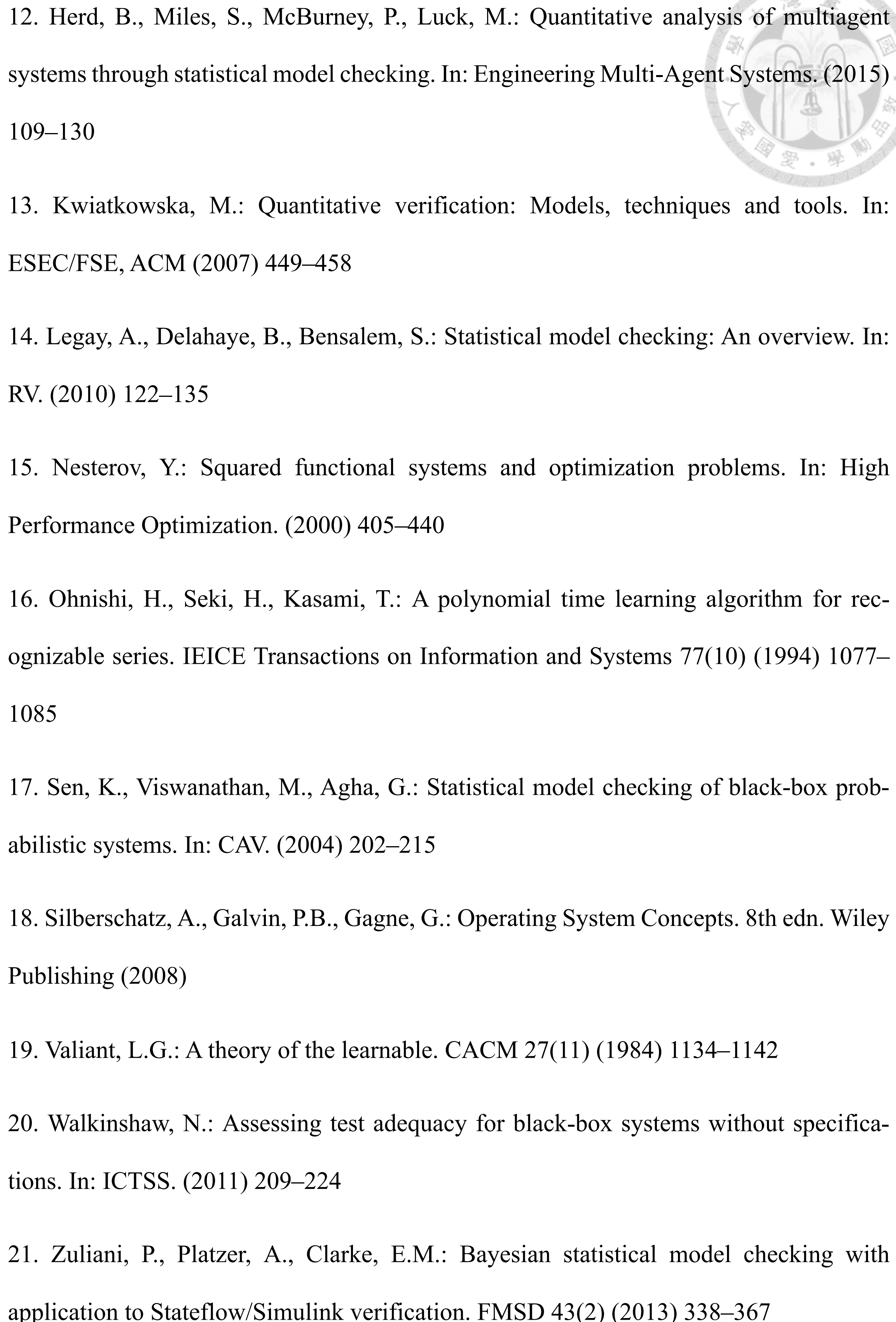
8. Carlyle, J., Paz, A.: Realizations by stochastic finite automata. Journal of Com- puter and System Sciences 5(1) (1971) 26–40

9. Chen, Y.F., Hsieh, C., Lengál, O., Lii, T.J., Tsai, M.H., Wang, B.Y., Wang, F.: PAC learning-based verification and model synthesis. In: ICSE. (2016) 714–724

10. Fliess, M.: Matrices de Hankel. J. Math. Pures Appl 53(9) (1974) 197–222

11. Goldreich, O., Goldwasser, S., Ron, D.: Property testing and its connection to learning and approximation. JACM 45(4) (1998) 653–750

doi:10.6342/NTU201701657



12. Herd, B., Miles, S., McBurney, P., Luck, M.: Quantitative analysis of multiagent systems through statistical model checking. In: Engineering Multi-Agent Systems. (2015) 109–130

13. Kwiatkowska, M.: Quantitative verification: Models, techniques and tools. In: ESEC/FSE, ACM (2007) 449–458

14. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: RV. (2010) 122–135

15. Nesterov, Y.: Squared functional systems and optimization problems. In: High Performance Optimization. (2000) 405–440

16. Ohnishi, H., Seki, H., Kasami, T.: A polynomial time learning algorithm for recognizable series. IEICE Transactions on Information and Systems 77(10) (1994) 1077–1085

17. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: CAV. (2004) 202–215

18. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts. 8th edn. Wiley Publishing (2008)

19. Valiant, L.G.: A theory of the learnable. CACM 27(11) (1984) 1134–1142

20. Walkinshaw, N.: Assessing test adequacy for black-box systems without specifications. In: ICTSS. (2011) 209–224

21. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to Stateflow/Simulink verification. FMSD 43(2) (2013) 338–367

doi:10.6342/NTU201701657