Supporting system design with automaton learning algorithms

BACHELOR'S THESIS

Author
Áron Barcsa-Szabó

Advisor
Rebeka Farkas
Dr. András Vörös

November 27, 2019

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Barcsa-Szabó Áron*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2019. november 27.

_____

Barcsa-Szabó Áron
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamos-
mérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és dip-
lomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-
implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

Chapter 1

Introduction

1.1 Overview

1.2 Problem Statement

1.3 Objective

1.4 Contribution

1.5 Outline

Chapter 2 provides an outline of the necessary technical background.

Chapter 2

Background

This chapter provides some theoretical background of the contributions presented in this thesis. First of all, the necessary basics of formal language and automaton theory are introduced, afterwards we discuss automaton learning algorithms.

2.1 Basics of automaton theory

First, we introduce the fundamentals of formal language theory, which are essential in order to understand automaton theory. Atomic elements of formal languages are alphabets, characters and words.

Definition 1 (Alphabet). Let Σ be a finite, non-empty set. Σ is an Alphabet, its elements are symbols or characters. ∎

Definition 2 (Word). If Σ is an alphabet, then any finite sequence comprised of the symbols of Σ are words (or Strings). Σ^n represents the set of every n length word in Σ. The set of every word under an alphabet, formally $\bigcup_{n>0} \Sigma^n$ is denoted by Σ^*. The empty word is denoted by ϵ. ∎

Words can be constructed using other words, the following definition defines these relations.

Definition 3 (Prefixes, Substrings and Suffixes). If $w = uvs$, where $w, u, v, s \in \Sigma^*$, u is the prefix, v is the substring, and s is the suffix of w. ∎

Using these atomic elements of formal language theory, the definition of formal language theory can be seen as follows.

Definition 4 (Formal Language). An arbitrary set of words under an Alphabet Σ is a Language. Formally: $L \subseteq \Sigma^*$. ∎

Definition 5 (Prefix-closure). Let $L \subseteq \Sigma^*$ and $L' = \{u \in \Sigma^*, v \in \Sigma^* : uv \in L\}$. In other words, L' is the set containing all the prefixes of every word of L. L is prefix-closed if $L = L'$. ∎

We will expand on formal languages more once we have a basic understanding of automatons.

Informally, Automatons, or automata are mathematical constructs which are able to determine if a sequence of inputs should be accepted or rejected. A bit more precisely, automata consist of states and is always in one of them. Starting from an initial state, based on the inputs received, the automaton changes, "moves" between states. Essentially, for every one of the inputs, based on the current state the automaton is in, it determines whether to keep, or change its current state. In order to determine if an input sequence should be accepted or not, some states are special, accepting states. If after processing a sequence of inputs, the final state of the automaton is accepting, the input sequence is accepted. If not, the input is rejected.

One of the most simple automaton is the Deterministic Finite Automaton.

Definition 6 (Deterministic Finite Automaton). A Deterministic Finite Automaton is a Tuple of $DFA = (S, s_0, \Sigma, \delta, F)$, where:

- S is a finite, non-epty set containing the states of the automaton,

- $s_0 \in S$ is the initial state,

- Σ is a finite Alphabet,

- $\delta : S \times \Sigma \to S$ is a transition function,

- $F \subseteq S$ is a set of the accepting states of the automaton. ∎

An example of a DFA (Deterministic Finite Automaton) from[8] can be seen in Example 1.

Example 1. *See Fig. 2.1. This example has four states, $S = \{q_0, q_1, q_2, q_3\}$ (hence $|S| = 4$). The initial state is marked by the start arrow, so $s_0 = q_0$. The alphabet can be inferred as $\Sigma = \{a, b\}$ because of the deterministic in Deterministic Finite Automaton, meaning every state must deterministically know what input causes what action. This means, that every state must have every member of the alphabet listed in its transitions. Transitions are visualized as $q_0 \xrightarrow{a} q_1$ given by the transition function (in this example) $\delta(q_0, a) = q_1$. The whole of the transition function in a table form can be seen in Table 2.1. Finally, the accepting states, or in this case, accepting state of the automaton is $F = \{q_3\}$.*

When talking about automata, it is important to discuss runs. A run of an automaton is to test for a certain input (word), if it is accepted or rejected. See Example 2.

Example 2. *A run of Fig. 2.1 with an input of $\{a, a, a\}$ would (following the transition function) end in state q_3 meaning the input is accepted. A rejected input could be $\{a, b, b\}$, which would stop at state q_1, a non-accepting state. On deeper examination, one can see, that this automaton only accepts runs with inputs containing $4i + 3a$.*
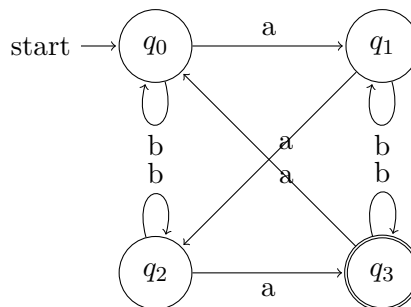

Figure 2.1. A simple DFA from [4].

3

δ	q_0	q_1	q_2	q_3
a	q_1	q_2	q_3	q_0
b	q_0	q_1	q_2	q_3

Table 2.1. The transition function of the automaton seen in Fig. 2.1

DFAs are great to model system behavior based on inputs, but in order to work with reactive systems, we also need to handle outputs. Mealy machines are automata designed to do just this, while becoming more complicated with regard to accepting inputs.

Definition 7 (Mealy machine). A Mealy machine or Mealy automaton is a Tuple of $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$, where:

- S is a finite, non-empty set containing the states of the automaton,

- $s_0 \in S$ is the initial state,

- Σ is the input alphabet of the automaton,

- Ω is the output alphabet of the automaton,

- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function and

- $\lambda : Q \times \Sigma \rightarrow \Omega$ is the output function. ▪

Mealy machines can be regarded as deterministic finite automata over the union of the input alphabet and an output alphabet with just one rejection state, which is a sink, or more elegantly, with a partially defined transition relation.

An example of a deterministic Mealy machine can be seen in Example 3.

Example 3. *An example of a deterministic Mealy machine can be seen in Fig. 2.2. The formal definition of the automaton can be seen below.*

- $S = \{a, b, c, d, d', e, f\}$

- $s_0 = a$

- $\Sigma = \{water, pod, button, clean\}$

- $\Omega = \{\checkmark, ☕, \star\}$

The transitions, as seen in Fig. 2.2 are visualized as $s_0 \xrightarrow{input/output} s_1$, which denotes the machine moving from state s_0 to state s_1 on the specified input, while causing the specified output. Also, some simplifications are done, e.g. in this transition: $d \xrightarrow{\{water,pod\}/\checkmark} d$ we see a visual simplification of having both transitions merged to one arrow, this is only for visual convenience. Fig. 2.2 is also a great example of sinks, as seen in state f, the machine accepts anything, and never changes. This is a variation of the accepting state seen in DFAs.
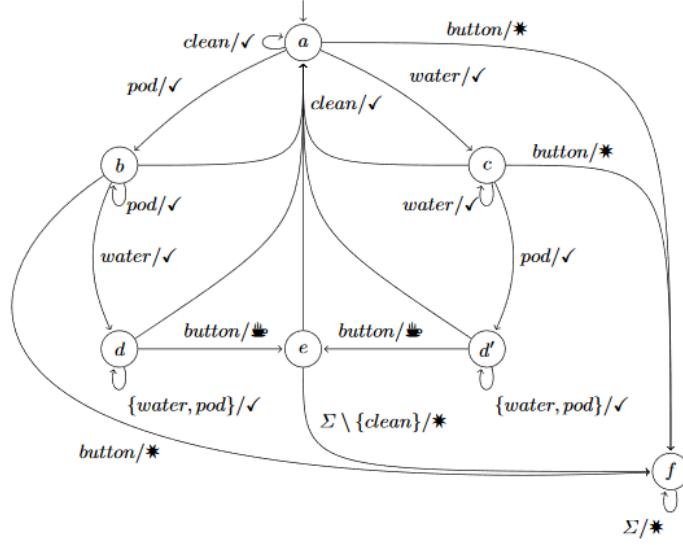
Figure 2.2. Mealy machine representing the functionality of a coffee machine.[8]

Since automatons deal with alphabets, formal language theory is essential not only to work with them, but to build them efficiently. Often automata are used to model and test real-life systems. Naturally, questions of efficiency, correctness arise, which is why we will next expand on the relations of automatons and formal languages.

Definition 8 (Recognized language of automata). The language $L \subseteq \Sigma$ containing all the accepted words by an automaton M is called the recognized language of the automaton. It is denoted by $L(M) = L$. ∎

Definition 9 (Regular language). A formal language L is regular, iff there exists a Deterministic Finite Automaton M, for which $L(M) = L$, in other words, iff there exists a DFA with the recognized language of L. ∎

We will now introduce a semantic helper δ^* for both DFAs and Mealy machines. δ^* is an extension of the δ transition function, as $\delta^* : S \times \Sigma^* \to S$ defined by $\delta^*(s, \epsilon) = s$ and $\delta^*(s, \alpha w) = \delta^*(\delta(s, \alpha), w)$, essentially giving us the state of the automaton after running an input sequence from a specified state.

Definition 10 (Myhill-Nerode relation). A DFA $M = (S, s_0, \Sigma, \delta, F)$ induces the following equivalence relation $\equiv_M$ on Σ^* (when $L(M) = \Sigma$):
    $x \equiv_M y \iff \delta^*(s, x) = \delta^*(s, y)$
where $x, y \in \Sigma^*$. This means, that x and y are equivalent with respect to $\equiv_M$.[5]. ∎

The Myhill-Nerode relation is an equivalence relation, meaning $x \equiv_M y$ is reflexive, symmetric and transitive. Also, the following properties apply to it[5].

- Right congruence: $\forall x, y \in \Sigma^* : (x \equiv_M y \implies \forall a \in \Sigma : xa \equiv_M ya)$
  also, by induction, this can be extended to:
  $\forall x, y \in \Sigma^* : (x \equiv_M y \implies \forall w \in \Sigma^* : xw \equiv_M yw)$.

- It respects membership wrt. R:
  $\forall x, y \in \Sigma^* : x \equiv_M y \implies (x \in R \iff y \in R)$

5

- $\equiv_M$ is of finite index, has finitely many equivalency classes. Since for every state $s \in S$, the sequences which end up in s are in the same equivalence class, the number of these classes is exactly $|S|$, which is a finite set.

Using this relation, we can introduce the Myhill-Nerode theorem, which neatly ties together the previous definitions.

Theorem 1 (Myhill-Nerode theorem[5][7]). Let $L \subseteq \Sigma^*$. The following three statements are equivalent:

- L is regular.

- there exists a Myhill-Nerode relation for L.

- the relation $\equiv_L$ is of finite index.

For proof, see [5][7]. $\blacksquare$

Next, we will introduce the same topics to Mealy machines, which are a bit more complex in this regard. In order to do this, we need to introduce a semantic helper similar to δ^*, but considering the output function of Mealy machines. $\lambda^* : S \times \Sigma^* \to \Omega$, defined by $\lambda^*(s, \epsilon) = \varnothing$ and $\lambda^*(s, w\alpha) = \lambda(\delta^*(s, w), \alpha)$.

When monitoring the behavior of Mealy machines, one of the most important metrics given an input is not whether it is accepted or rejected (as it would be with a DFA), but rather what specific output was caused by an input. The behavior of a Mealy machine, a specific run of it, has a pattern of $i_1, o_1, i_2, o_2, .., i_n, o_n$, where i are inputs and o are outputs. But in order to characterize these runs, we actually do not need every output from this pattern, we only need the final one. This means, that a $[\![M]\!]$: $\Sigma^* \to \Omega$ semantic functional fully captures the behavioral semantics of Mealy machines. We define $[\![M]\!]$: $\Sigma^* \to \Omega$ as $[\![M]\!]$(w) $= \lambda^*(s_0, w)$. Informally,the $[\![M]\!]$ functional accepts a set of inputs, and returns the last output given after running them from the initial state of the automaton.

Example 4. *Given the Mealy machine M$_{coffeemachine}$ in Fig. 2.2, the runs:*
$<$clean, ✓ $>$,
$<$pod water button, ☕$>$
are in $[\![M_{coffeemachine}]\!]$, since these input words cause these outputs, while the runs
$<$clean, ☕$>$ and
$<$water button button, ✓ $>$
are not, since these input sequences do not produce those outputs.

Similarly to the Myhill-Nerod relations in DFAs, we can introduce an equivalence relation over the $P : \Sigma^* \to \Omega$ functional. P being an abstraction of $[\![M]\!]$, since the latter only maps from initial states.

Definition 11 (Equivalence of words wrt. $\equiv_P$[8]). Given a Mealy machine $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$, two words, $u, u' \in \Sigma^*$ are equivalent with respect to $\equiv_P$:
$u \equiv_P u' \iff (\forall v \in \Sigma^* : P(s, uv) = P(s, u'v))$.
We write [u] to denote the equivalence class of u wrt. $\equiv_P$. $\blacksquare$

This definition is more along the lines of the right congruence property observed in the Myhill-Nerode relations, the original ($u \equiv_P u' \iff P(s, u) = P(s, u')$) format would of course still hold, in this case, $v = \epsilon$.

Example 5. *Taking Fig. 2.2 as an example, the following words are equivalent wrt.* $\equiv_{[\![M]\!]}$:

$$\begin{array}{ll} & \textit{water, pod} \\ \equiv_{[\![M]\!]} & \textit{water, water, pod} \\ \equiv_{[\![M]\!]} & \textit{pod pod water.} \end{array}$$

The first two of these are straightforward, since both lead to the same state, d', more interesting is the pod pod water input, which ends in state d. Observably, state d and d' wrt. outputs operate exactly the same regardless of continuation, this is why the equivalence holds.

Theorem 2 (Characterization theorem[8]). Iff mapping $P : \Sigma^* \to \Omega \equiv_P$ has finitely many equivalence classes, there exists a Mealy machine M, for which P is a semantic functional.

Proof($\Longleftarrow$): As seen in the case of the Myhill-Nerode finite index property for DFAs, same states in Mealy machines will obviously be in same equivalence classes. This implies, that the number of classes in (or in other words, the index of) $\equiv_P$ is at most the number of states the Mealy machine contains, which is finite by definition.

Proof($\Longrightarrow$): Consider the following Mealy machine: $M_P = (S, s_0, \Sigma, \Omega, \delta, \lambda)$:

    -S is given by the equivalence classes of $\equiv_P$.

    -s_0 is given by $[\epsilon]$.

    -δ is defined by $\delta([u], \alpha) = [u\alpha]$.

    -λ is defined by $\lambda([u], \alpha) = o$, where $P(u\alpha) = o$.

A Mealy machine constructed this way fulfills what the theorem states, P is a semantic functional of it, in other words, $[\![M]\!] = P$. ∎

With this theorem, we can define regularity for mappings $P : \Sigma^* \to \Omega$. We call a $P : \Sigma^* \to \Omega$ mapping regular, iff there exists a corresponding Mealy machine for which $[\![M]\!] = P$, or equivalently, if P has a finite number of equivalence classes, analogously to the previously seen "classical" regularity.

The introduction of regularity helps us in the construction of automata, specifically, the construction of canonical automata.
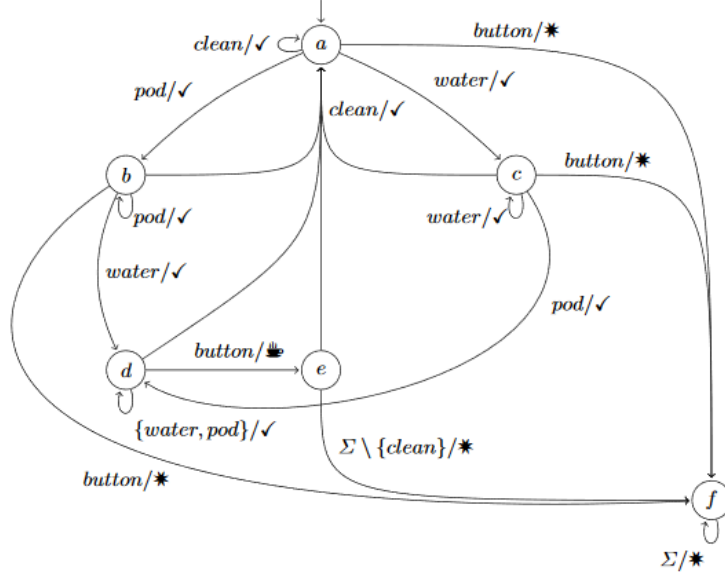
Figure 2.3. Minimal version of the Mealy machine seen in 2.2

Definition 12 (Canonical automaton (Minimal automaton)). An automaton M is canonical (i.e. minimal) iff:

- every state is reachable: $\forall s \in S : \exists w \in \Sigma^* : \delta^*(s_0, w) = s$,

- all states are pairwisely separable, in other words behaviorally distinguishable. For Mealy machines, this is formalized as: $\forall s_1, s_2 \in S : \exists w \in \Sigma^* : \lambda(s_1, w) \neq \lambda(s_2, w)$ ▪

Constructing automata to be canonical, especially in the case of Mealy machines is important with regards to efficiency and is the backbone of automaton learning. We will steer to algorithm theory that does this, before actually covering the topic of automaton learning itself. The next proposition comes straightforward from the previously presented characterization theorem.

Proposition (Bounded reachability[8]): Every state of a minimal Mealy machine with n states has an access sequence, i.e., a path from the initial state to this state, of length at most n-1. Every transition of this Mealy machine can be covered by a sequence of length at most n from the initial state.

The process of constructing automata will use the concept of partition refinement. It works based on distinguishing suffixes, suffixes of words which mark, witness the difference between two access sequences. We'll introduce the following notion to formalize this.

Definition 13 (k-distinguishability[8]). Two states, $s, s' \in S$ are k-distinguishable iff there is a word $w \in \Sigma^*$ of length k or shorter, for which $\lambda^*(s, w) \neq \lambda^*(s', w)$. ▪

Definition 14 (exact k-distinguishability). Two states, $s, s' \in S$ are exact k-distinguishable, denoted by $k^=$ iff s and s' are k-distinguishable, but not (k-1)-distinguishable ▪

Essentially, if two states, s and s' are k-distinguishable, then when processing the same input sequence, from some suffix of the word w with length at most k, they will produce differing outputs. Using this, we can observe, that whenever two states, $s_1, s_2 \in S$ are

(k+1)-distinguishable, then they each have an α-successor, s_1' and s_2' for some $\alpha \in \Sigma$, such that s_1' and s_2' are k-distinguishable. This suggests, that:

- no states are 0-distinguishable and

- two states s_1 and s2 are (k+1)-distinguishable iff there exists an input symbol $\alpha \in \Sigma$, such that $\lambda(s_1, \alpha) \neq \lambda(s_2, \alpha)$ or $\delta(s_1, \alpha)$ and $\delta(s_2, \alpha)$ are k-distinguishable.[8]

This way, if we have an automaton M, we can construct its minimal version, by iteratively computing k-distinguishability for increasing k, until stability, that is until the set of exactly k-distinguishable states is empty.

Example 6. *Given the Mealy machine seen in Fig.2.2, we can use k-distinguishability to refine its partitions. The initial state, the initial partition would be:*
$P_1 = \{a, b, c\}, \{d, d'\}, \{e\}, \{f\}$
since when k=1, a, b and c are not 1-distinguishable, but d and d' separate on the behavior of the button input, while e and f are separated by the suffix clean. Let's see the k=2 scenario.
$P_2 = \{a\}, \{b\}, \{c\}, \{d, d'\}, \{e\}, \{f\}$
Here, water and pod separate a, b and c, while d and d' can still no longer be separated. If observed, even if k is increased, d and d' can not be refined. This means, that they are indistinguishable, they can be merged together without altering behavior. This shows the process of acquiring the minimal machine seen in Fig. 2.3.

The process explained in the above example is partition refinement, the exact algorithm and proof of its validity can be seen in [8], it is a version of the minimization algorithm for DFAs proposed by Hopcroft[2].

We will define one last relation which will help us in the next section to compare automata minimization and automata learning.

Let $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$ and $M = (S', s_0', \Sigma, \Omega, \delta', \lambda')$ be two Mealy machines with shared alphabets. We call a surjective function $f_k : S \to S'$ existential k-epimorphism between M and M', if for all $s' \in S', s \in S$ where $f_k(s) = s'$ and with any $\alpha \in \Sigma$, we have: $f_k(\delta(s, \alpha)) = \delta'(s', \alpha)$, and all states, that are mapped by f_k to the same state of M' are not k-distinguishable. It is straightforward to establish that all intermediate models arising during the partition refinement process are images of the considered Mealy machine under a k-epimorphism, where k is the number of times all transitions have been investigated.[8] Essentially this establishes P_1 and P_2 from Example 6 as images of the Mealy machine seen in Fig. 3 under k-epimorphisms where k=1 and k=2 respectively.

Active automaton learning algorithms operate in a very similar way, but they are not the same as minimization algorithms, since they do not have access to the automatons they are learning. We will extend upon this in the next section.

2.2 Automaton learning

Automaton learning is modeling a system without having specific knowledge of its the internal behavior. To accomplish this, a model needs to be inferred by observing the external behavior of the system. This learned model is, as the name suggests, an automaton.
Formally: Active automata learning is concerned with the problem of inferring an

automaton model for an unknown formal language L over some alphabet Σ[3].

In order to monitor a system, a way of access to its behavioral information is required. There are two approaches, which separate the two types of automaton learning as well.

Passive automaton learning   In case of passive automaton learning, the gathering of information is not part of the learning process, but rather a prerequisite to it. The learning is performed on a pre-gathered positive an/or negative example set of the systems behavior. In passive automaton learning, the success of the process is determined not only by the efficiency of the algorithm, but the methodology and time used to gather the data.

Active automaton learning   In case of active automaton learning, the behavioral infromation is gathered by the learning algorithm in an "active" way via queries. In order to accomplish this, learning is separated to two components: the learner, which learns, and the teacher, which can answer questions about the system under learning.

Active automaton learning follows the MAT, or the Minimally Adequate Teacher model proposed by Dana Angluin[1]. It separates the algorithm to a learner and a teacher, where the teacher can only answer the minimally adequate questions needed to learn the system. These two questions, or queries are are follows:

Membership query   Given a $w \in \Sigma^*$ word, the query return the $o \in \Omega$ output corresponding to it, treating the word as a string of inputs. We write $mq(w) = o$ to denote that executing the query w on the system under learning (SUL) leads to the output o: $[\![SUL]\!](w) = o$ or $\lambda^*(s_0, w) = o$.

Equivalence query   Given a hypothesis automaton M, the query tries to determine if the hypothesis is behaviorally equivalent to the SUL, and if not, finding the diverging behavior, and return with the example of it. We write $eq(H) = c$, where $c \in \Sigma^*$, to denote an equivalence query on hypothesis H, returning a counterexample c. The counter example provided is the sequence of inputs for which the output of system under learning and the output of the hypothesis differ: $[\![H]\!](c) \neq mq(c)$.

The learner component uses membership queries to construct a hypothesis automaton, then refines this hypothesis by the counterexamples provided by equivalence queries. Once counterexamples can not be found this way, the learners hypothesis is behaviorally equivalent to the SUL, the learning can terminate and the output of the learning is the current hypothesis.
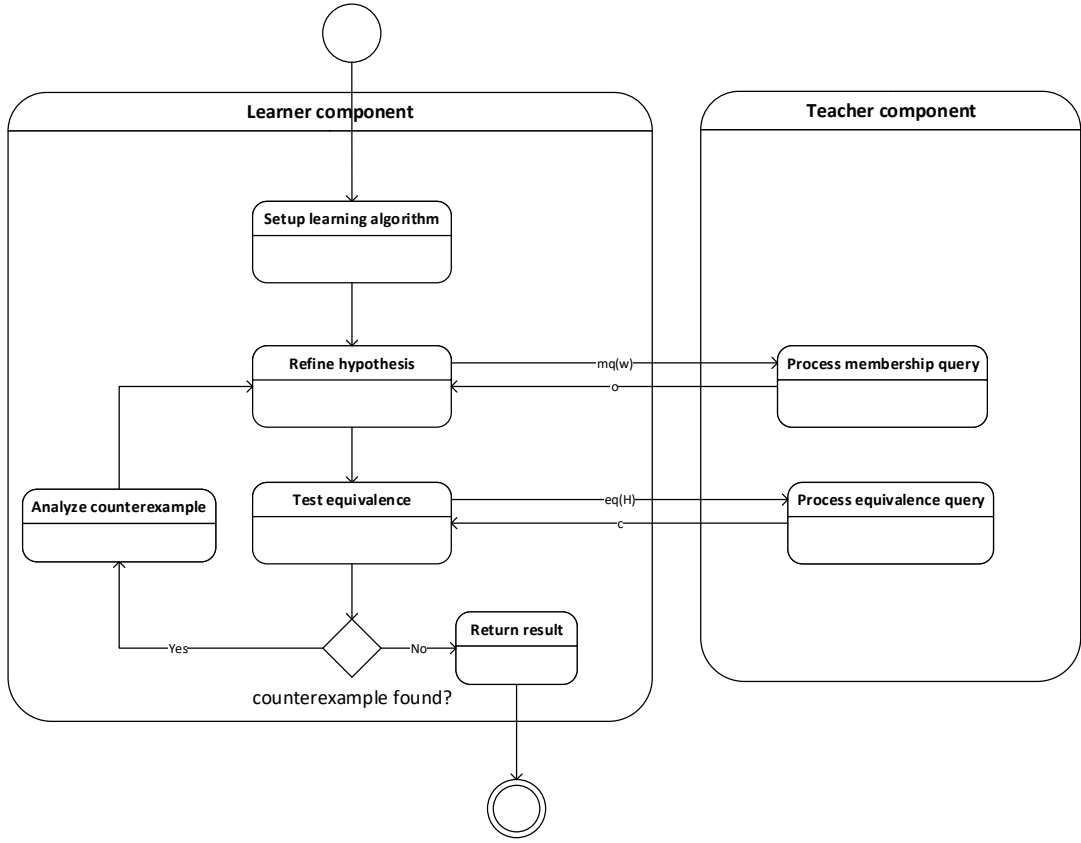
Figure 2.4. Abstract model of active automata learning algorithms

As seen on Fig. 2.4, the learning proceeds in rounds, generating and refining hypothesis models by exploring the SUL via membership queries. As the equivalence checks produce counterexamples, the next round of this hypothesis refinement is steered by the counterexamples produced.

Using an analogous strategy to the minimization of automatons seen in the previous section, starting only with a one state hypothesis automaton, we explore over all words in the alphabet in order to refine and extend this hypothesis. Here, there is a dual way of characterizing (and distinguishing) between states[8]:

- By words reaching them. A prefix-closed set of S_p words, reaching each state exactly once, defines a spanning tree of the automaton. This characterization aims at providing exactly one representative element from each class of $\equiv_P$ on the SUL. Active learning algorithms incrementally construct such a set S_p.
  This prefix-closedness is necessary for S_p to be a "spanning tree" of the Mealy machine. Extending S_p with all the one-letter continuations of words in S_p will result in the tree covering all the transitions of the Mealy machine. L_p will denote all the one-letter continuations that are not already contained in S_p.

- By their future behavior with respect to an increasing vector of words of Σ^*. This vector $< d_1, d_2, ..., d_k >$ will be denoted by D, and contains the "distinguishing suffixes". The corresponding future behavior of a state, here given in terms of its access

sequence $u \in S_p$, is the output vector$< mq(u * d_1), ..., mq(u * d_k) >\in \Omega^k$, which leads to an upper approximation of the classes of $\equiv_{[\![SUL]\!]}$. Active learning incrementally refines this approximation by extending the vector until the approximation is precise.

While the second characterization defines the states of the automaton, where each output vector corresponds to one state, the spanning tree on L_p is used to determine the transitions of these states. In order to characterize the relation between the SUL $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$ and the hypothesis model $M' = (S', s_0', \Sigma, \Omega, \delta', \lambda')$ (note, that M and M' only share alphabets) let $D \subseteq \Sigma^*$. We call a surjective function $f_D : S \to S'$ existential D-epimorphism (surjective homomorphism) between M and M' if, for all $s' \in S'$ there exists an $s \in S$ with $f_D(s) = s'$ such that for all $\alpha \in \Sigma$ and all $d \in D$: $f_D(\delta(s, \alpha)) = \delta'(s', \alpha)$, and $\lambda^*(s, d) = \lambda^*(s', d)$.
Note, that active learning deals with canonical Mealy machines, in other words, the canonical form of SUL, and not, the perhaps much larger Mealy machine of SUL itself.

Since active learning algorithms maintain an incrementally growing extended spanning tree for $H = (S_H, h_0, \Sigma, \Omega, \delta_H, \lambda_H)$, i.e., a prefix-closed set of words reaching all its states and covering all transitions, it is straightforward to establish that these hypothesis models are images of the canonical version of SUL under a canonical existential D-epimorphism, where D is the set of distinctive futures underlying the hypothesis construction[8]

- define $f_D : S_{SUL} \to S_H$ by $f_D(s) = h$ as following: if $\exists w \in S_p \cup L_p$, where $\delta(s_0, w) = s$, then $h = \delta_H(h_0, w)$. Otherwise h may be chosen arbitrarily.

- It suffices to consider the states reached by words in the spanning tree to establish the defining properties of f_D. This straightforwardly yields:

  - $f_D(\delta(s, \alpha)) = \delta_H(h, \alpha)$ for all $\alpha \in \Sigma$, which reflects the characterization from below.
  - $\lambda^*(s, d) = \lambda_H^*(h, d)$ for all $d \in D$, which follows from the maintained characterization from above.[8]

In basic logic, D-epimorphisms and k-epimorphisms do not differ, they both deal with establishing constructed models being images of the model they are based on. D-epimorphisms could replace k-epimorphisms where $D = \Sigma^k$, it can be suggested, that there is no need to differentiate. However, there is in important difference of complexity between the two. While k-distinguishability supports polynomial time, black-box systems do not. Also, the "existential" in existential D-epimorphism is important: f_D must deal with unknown states, ones that haven't been encountered yet. This implies that characterization can only be valid for already encountered states.

Active learning algorithms can be proven correct using the following three-step pattern:

- Invariance: The number of states of each hypothesis has an upper bound of $\equiv_{[\![SUL]\!]}$.

- Progress: Before the final partition is reached, an equivalence query will provide a counterexample, where an input word leads to a different output on the SUL and on the hypothesis. This difference can only be resolved by splitting at least one state, which increases the state count.

- Termination: The refinement terminates after at most the index of $\equiv_{[\![SUL]\!]}$ many steps, caused directly by the described invariance and progress properties.

The following subsection introduces the first active automaton learning algorithm this thesis covers.

2.2.1 Direct Hypothesis Construction[6]

The Direct Hypothesis Construction algorithm, which hypothesis construction can be seen in Algorithm 1 follows the idea of the breath-first search of graph theory, it constructs the hypothesis using a queue of states, which is initialized with the states of the spanning tree to be maintained. Explored states are removed from this queue, while the discovered successors are enqueued, if they are provably new states. The algorithm starts with a one-state hypothesis, including only the initial state, reached by ϵ and $D = \Sigma$. It then tries to complete the hypothesis, for every state we determine the behavior of the state under D, we will call this behavior the extended signature of said state. States with a new extended signatures are provably new states, so to guarantee further investigation, we enqueue all their successors. Initially, $D = \Sigma$, so only the $1^=$-distinguishable states are revealed during the first iteration. This is extended straightforwardly to comprise a prefix closed set of access sequences. [8][6]

Algorithm 1: Hypothesis construction of the Direct Hypothesis Construction algorithm as seen in [8].

Input: S_p: a set of access sequences, D: a set of suffixes, an input alphabet Σ
Output: A Mealy machine $H = (S, s_0, \Sigma, \Omega, \delta, \lambda)$

1 initialize hypothesis H, create a state for all elements of S_p
2 initialize a queue Q with the states of H
3 **while** *Q is not empty* **do**
4 $\quad$ s = dequeue state from Q
5 $\quad$ u = access sequence from s_0 to s
6 $\quad$ **for** $d \in D$ **do**
7 $\quad\quad$ o = mq(ud)
8 $\quad\quad$ set $\lambda(s, d) = o$
9 $\quad$ **end**
10 $\quad$ **if** *exists an $s' \in S$, where the output signature of s' is the same as s* **then**
11 $\quad\quad$ reroute transitions of s to s' in H
12 $\quad\quad$ remove s from H
13 $\quad$ **else**
14 $\quad\quad$ create and enqueue successors of s for every input in Σ into Q, if not already in S_p
15 $\quad$ **end**
16 **end**
17 Remove entries of $D \setminus \Sigma$ from λ
18 **return** *H*

After the run of the Hypothesis construction seen in Algorithm 1, the output automaton H is used in an equivalence query eq(H) = c, to find if a counterexample c exists. If no counteraxamle can be found, the learning terminates, H is the learned automaton. Else, if a counterexample c is found, for which $\lambda_H(s_0, c) \neq mq(c)$, c is used to enlarge the suffixes in D and a new iteration of Algorithm 1 begins, using the now extended set D and all the access sequences found in the previous iteration, the current spanning tree S_p.

While the DHC algorithm is a straightforward implementation of active automata learning, it struggles with time complexity, it terminates after at most $n^3mk + n^2k^2$ membership

queries, and n equivalence queries, where n is the number of states in the final hypothesis, k is the longest set of inputs, and m is the length of the longest counterexample. This of course is far from efficient, the next subsection deals with a far more optimal algorithm, the TTT algorithm.

2.2.2 The TTT algorithm[4]

As seen in the Direct Hypothesis Construction algorithm, a large number membership queries are required in order to learn an automaton, causing a bottleneck on the scalability of the algorithm. The main cause of this complexity issue is the way counterexamples are treated. In order to ensure every state is properly found, *all* suffixes of the counterexample are added to the algorithms structure (denoted D in the previous subsection), which, in the next iteration of the hypothesis construction, has a huge impact on the number of membership queries. TTT resolves this inefficiency by fine-tuning exactly what needs to be further investigated by membership queries.
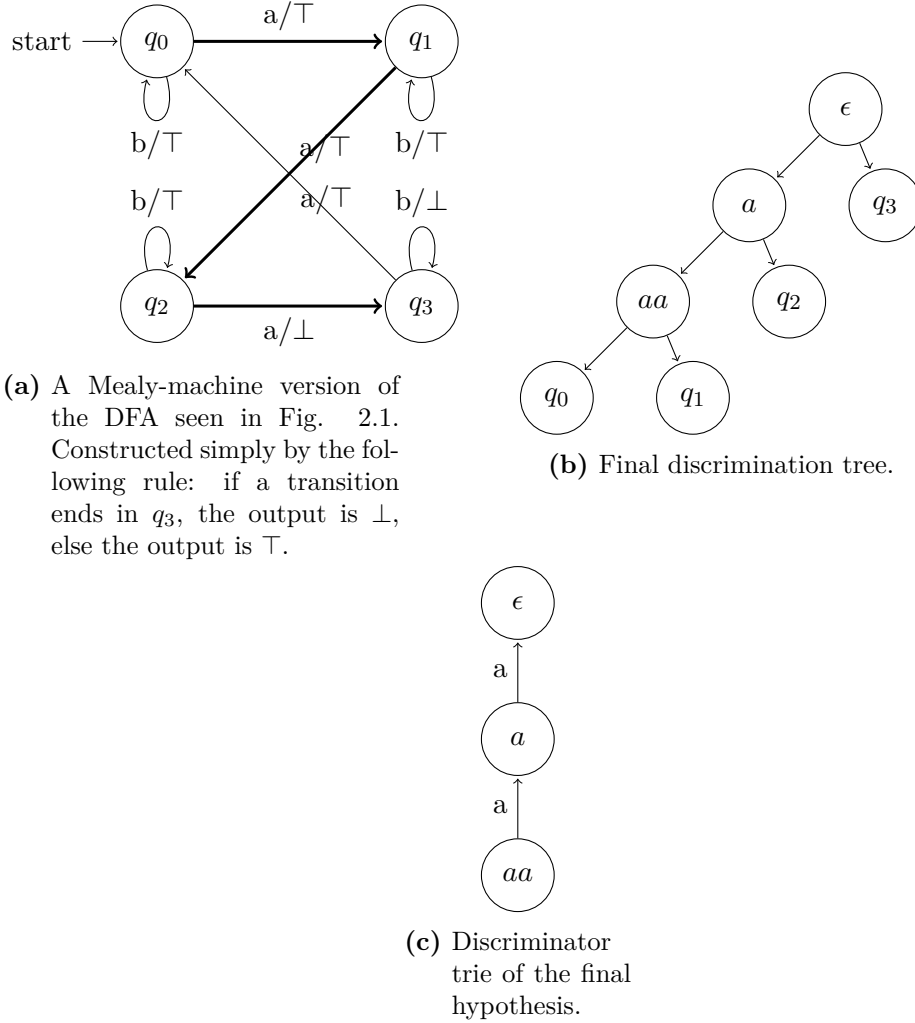

(a) A Mealy-machine version of the DFA seen in Fig. 2.1. Constructed simply by the following rule: if a transition ends in q_3, the output is $\bot$, else the output is $\top$.

(b) Final discrimination tree.

(c) Discriminator trie of the final hypothesis.

Figure 2.5. The three namesake trees of the TTT algorithm, (a) being a spanning **T**ree, (b) a disctimination **T**ree, and (c) a discriminator **T**rie.

Example 7. *As an example, we'll use the automaton portrayed on Fig.2.5.a a Mealy machine version of the DFA previously seen in Fig.2.1. TTT keeps the notion of separating states by a S_p spanning tree, a prefix-closed set of words from Σ^*, this tree defines the access sequences of each state. The spanning tree S_p is indicated by bold transitions in Fig.2.5.a. States are separated by the notion of discriminators. The algorithm maintains a set of discriminator suffixes D, but uses them differently from the DHC algorithm. D in TTT is used by a n-ary tree, where $n = |\Omega|$, so in this example a binary tree. We cal this tree a discrimination tree (seen in Fig.2.5.b), which maintains information about which inputs (discriminators) separate states: for every distinct pair of states, a separator can be obtained by looking at the label of the lowest common ancestor of the corresponding leaves. This way, the label of inner nodes act as discriminators. These labels form a suffix-close set and are stored in a trie seen in Fig.2.5.c. Each node of the trie is a word, which can be constructed by going up the tree towards the root.*

Discriminator trees, as seen in the previous example, are rooted n-ary trees, where n is the size of the output alphabet. Leaves are labeled by states of the automaton, while inner states are labeled by discriminators (suffixes). When adding a new word to the tree T, we sift the word $w \in \Sigma^*$ into T by starting at the root, and for every inner node $v \in D$, we branch depending on the output of $\lambda^*(wu)$. This limits the number of membership queries to the height of the tree, while DHCs limit was $S_p \times D$.

Key steps of the TTT algorithm:

- **Hypothesis construction.** The initial state is initialized and membership queried for all inputs, the results of which are sifted into the discriminator tree T.

- **Hypothesis refinement.** If a counterexample is found, the corresponding state is split (analogously to the DHC algorithm), in the discriminator tree, the leaf corresponding to this new state is split by a temporary discriminator $v \in \Sigma$.

- **Hypothesis stabilization.** The hypothesis provided by the previous example might contradict information of the discrimination tree. This step searches for counterexamples between them (without the need of equivalence queries), and if any such counterexample exist, states are split accordingly.

- **Discriminator finalization.** TTT treats discriminators derived directly from counterexamples temporary, keeping track of the maximal sub-tree of the discrimination tree containing temporary discriminators. This sub-tree is split as temporary discriminators are replaced with final ones. Only when this finalization happens, is an inner node of the discrimination tree considered a member of the discriminator trie, in other words, when finalizing a temporary discriminator, it is added to the discriminator trie.

A more thorough view on the TTT algorithm can be seen here[4].

Acknowledgements

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

Bibliography

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987. ISSN 0890-5401. URL `https://doi.org/10.1016/0890-5401(87)90052-6`.

[2] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189 – 196. Academic Press, 1971. ISBN 978-0-12-417750-5. DOI: `https://doi.org/10.1016/B978-0-12-417750-5.50022-1`. URL `http://www.sciencedirect.com/science/article/pii/B9780124177505500221`.

[3] Falk Howar and Bernhard Steffen. *Active Automata Learning in Practice*, pages 123–148. Springer International Publishing, Cham, 2018. ISBN 978-3-319-96562-8. DOI: `10.1007/978-3-319-96562-8_5`. URL `https://doi.org/10.1007/978-3-319-96562-8_5`.

[4] Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 307–322, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11164-3.

[5] Dexter C. Kozen. *Myhill—Nerode Relations*, pages 89–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 1977. ISBN 978-3-642-85706-5. DOI: `10.1007/978-3-642-85706-5_16`. URL `https://doi.org/10.1007/978-3-642-85706-5_16`.

[6] Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. Automata learning with on-the-fly direct hypothesis construction. In Reiner Hähnle, Jens Knoop, Tiziana Margaria, Dietmar Schreiner, and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 248–260, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-34781-8.

[7] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. ISSN 00029939, 10886826. URL `http://www.jstor.org/stable/2033204`.

[8] Bernhard Steffen, Falk Howar, and Maik Merten. *Introduction to Active Automata Learning from a Practical Perspective*, pages 256–296. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-21455-4. DOI: `10.1007/978-3-642-21455-4_8`. URL `https://doi.org/10.1007/978-3-642-21455-4_8`.

Appendix