BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

COMPARING LANGUAGES AND REDUCING AUTOMATA USED IN NETWORK TRAFFIC FILTERING

POROVNÁVÁNÍ JAZYKŮ A REDUKCE AUTOMATŮ POUŽÍVANÝCH PŘI FILTRACI SÍŤOVÉHO PROVOZU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. VOJTĚCH HAVLENA**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Abstract

The focus of this thesis is the comparison of languages and the reduction of automata used in network traffic monitoring. In this work, several approaches for approximate (language non-preserving) reduction of automata and comparison of their languages are proposed. The reductions are based on either under-approximating the languages of automata by pruning their states, or over-approximating the language by introducing new self-loops (and pruning redundant states later). The proposed approximate reduction methods and the proposed probabilistic distance utilize information from a network traffic. Formal guarantees with respect to a model of network traffic, represented using a probabilistic automaton are provided. The methods were implemented and evaluated on automata used in network traffic filtering.

Abstrakt

Tato práce se zabývá porovnáváním jazyků automatů a redukcí automatů používaných při monitorování síťového provozu. Je navrženo několik přístupů pro přibližnou redukci automatů (nezachovávající jazyk) a přístup pro porovnávání jejich jazyků. Redukce jsou založeny na podaproximaci jazyka automatu, kdy dochází k odstraňování stavů nebo na nadaproximaci jazyka, kdy dochází k přidávání nových smyček (a odstranění zbytečných stavů později). Navržené metody pro přibližnou redukci a navržená pravděpodobnostní vzdálenost využívají informaci ze síťového provozu. Jsou poskytnuty formální záruky vzhledem k modelu síťového provozu, který je reprezentován pravděpodobnostním automatem. Metody byly implementovány a jejich vlastnosti byly ověřeny na automatech používaných pro filtrování síťového provozu.

Keywords

language distance, network traffic filtering, weighted automata, finite automata, approximate reduction

Klíčová slova

vzdálenost jazyků, filtrování síťového provozu, váhované automaty, konečné automaty, přibližná redukce

Reference

HAVLENA, Vojtěch. *Comparing Languages and Reducing Automata Used in Network Traffic Filtering.* Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Vojnar Tomáš.

Comparing Languages and Reducing Automata Used in Network Traffic Filtering

Declaration

Hereby I declare that this thesis was prepared as an original author's work under the supervision of prof. Ing. Tomáš Vojnar, Ph.D. The supplementary information was provided by Ing. Ondřej Lengál, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Vojtěch Havlena
May 19, 2017

Acknowledgements

I would like to thank my supervisor Tomáš Vojnar for his time and numerous advice to this thesis. Great thanks also belong to Ondra Lengál, Milan Češka, jr., and Dana Hliněná for their advices and ideas. I am proud to be their student.

Contents

Chapter 1

Introduction

The recent growth of cyber-crime, in particular intrusion into computer networks, has greatly increased the demand for systems detecting malicious network traffic. In such systems, regular expressions are often used to describe packets to be selected for further inspection since they are, e.g., suspicious of containing an attack, tunneled protocol, etc. Due to the increasing speed of networks, network traffic filtering cannot be implemented in software, and some hardware pre-filtering is needed. These hardware solutions implement finite automata that correspond to regular expressions. However, computing resources available in the HW accelerators are restricted, and so methods for reducing the size of the automata must be used.

The reduction may be based on the classical approach of determinizing and minimizing the automata. This, however, incurs a possibly exponential explosion in the size of the determinized automata. Alternatively, the HW accelerator can be based on nondeterministic finite automata. Such automata can be reduced using, e.g., various approaches based on quotienting with respect to a simulation equivalence [15, 3] and further techniques like those proposed in [25] and implemented in the RABIT and Reduce tool [1]. Still, even such reductions need not be sufficient. For example, within our collaboration with the group of accelerated network technologies at FIT BUT (ANT@FIT), which is world-renowned in the area of hardware accelerated processing of high-speed network traffic, we were given regular expressions that translate to NFAs having from units to tens of thousands of states. Classical determinization and minimization simply explodes on these automata. Techniques of [25] may reduce the automata, according to our experience, to about half of the number of states. However, for HW accelerated network cards, sizes of around a few thousand of states are desirable. For hardware network probes, it is even less.

To improve on the above situation, we propose a novel approach based on an approximate reduction of the automata. Note that, the approximate reduction may change the language of the automata which can, in theory, lead to both false positives and false negatives when classifying the network traffic. However, this may still be better than not being able to run any classification at all or than having to completely ignore some traffic patterns. Moreover, one can also aim at the approximate reductions that will solely over-approximate the language. This can then increase the amount of packets sent from a hardware filtering device to the subsequent final software classification, but no critical traffic needs to be lost this way. In addition, we hope that the reduction techniques—whose development we have started in this thesis—can be fine-tuned such that a significant reduction of the automata can be achieved without a significant increase of the traffic sent for the final classification in software. Our first experimental results confirm this hypothesis.

In more concrete terms, since this thesis deals with approximate reduction methods, we first study suitable techniques for comparing a similarity of languages. Such techniques are needed so that we can control the reduction in a systematic way. For this reason, we propose a distance that is expressed as the probability that a randomly chosen string belongs to the symmetric difference of the input languages. The random string is chosen with respect to a distribution, which is represented by a probabilistic automaton (PA). This PA is obtained by an analysis of a representative sample of packets that occur in the network flow. Hence, the learned PA gives us a compact and abstract model of the network traffic, i.e., a representation of the frequency of occurrence of various packets in the network. Moreover, we propose a way how to evaluate the distance using matrix power series and some other related techniques.

Subsequently, we propose several automata reductions that are specifically tailored for use in network traffic monitoring. These reductions are based on either under-approximating the languages of automata by pruning their states, or over-approximating the languages by introducing new self-loops (and pruning redundant states later). The reductions can be parameterized by the maximal error that is allowed, which is given with respect to the desired distance between the language of the input automaton and the language of the reduced automaton. This parameterization allows the user to fine-tune the ratio between the achieved reduction and the error according to the concrete needs.

The proposed techniques were implemented and evaluated on a dataset provided by the ANT@FIT group. We have obtained highly promising results showing the potential of the proposed approach which can, moreover, be further improved in many ways as discussed at the end of the thesis.

The thesis is organized as follows. Chapter 2 serves as an introduction to the mathematical background of the studied techniques, finite automata theory, and weighted automata theory. In Chapter 3, we describe the use of finite automata in network traffic filtering. In Chapter 4, we introduce algorithms for computing the proposed distance between the languages of automata. Further, in Chapter 5, we describe the proposed techniques for the approximate reduction of automata. Chapter 6 then provides a description of a prototype tool for the approximate reduction that has been implemented. In Chapter 7, we focus on an experimental evaluation of the proposed techniques. Finally, in Chapter 8, we summarize the obtained results and propose possible future steps in the given area.

Chapter 2

Preliminaries

This opening chapter contains fundamentals used in the rest of this thesis. First, we introduce linear algebra background that we will later use for a computing of the proposed distance for comparing languages and for a computing of state labels for the approximate reduction of automata. Further, we recall distance functions and formal languages, and the last part is devoted to finite and weighted automata.

2.1 Linear Algebra Background

This section is based on [14, Chapter 9] and [26, Chapter 8]. We denote vectors as $\boldsymbol{\alpha}$ and matrices as $\mathbf{A}$. For a vector $\boldsymbol{\alpha}$ of l elements and an $m \times n$ matrix $\mathbf{A}$ (of m rows and n columns), we use the following functions: $\mathit{len}(\boldsymbol{\alpha}) = l, \mathit{rows}(\mathbf{A}) = m$, and $\mathit{cols}(\mathbf{A}) = n$. We define $\boldsymbol{\alpha}[i]$ for $1 \leq i \leq \mathit{len}(\boldsymbol{\alpha})$ to be the i-th element of vector $\boldsymbol{\alpha}$, and $\mathbf{A}\ [i,j]$ for $1 \leq i \leq \mathit{rows}(\mathbf{A})$ and $1 \leq j \leq \mathit{cols}(\mathbf{A})$ to be the element of $\mathbf{A}$ in row i and column j.

We use $[x, y]$ to denote the closed *interval* of real numbers from x to y, that is, $[x, y] = \{z \in \mathbb{R} \mid x \leq z \leq y\}$. Further, we use $\mathbb{R}_{\geq 0}$ to denote the set of all nonnegative real numbers (including zero), and $\mathbb{C}$ to denote the set of all complex numbers.

We use $\mathbf{I}$ to denote the *identity matrix*, $\mathbf{0}$ to denote the *zero matrix*, $\mathbf{A}^\top$ to denote the *transpose* of $\mathbf{A}$, and $\mathbf{A}^{-1}$ to denote the *inverse* of $\mathbf{A}$. An element $\lambda \in \mathbb{C}$ is an *eigenvalue* of a matrix $\mathbf{A}\ \in \mathbb{R}^{n \times n}$ if there exists a nonzero vector (i.e., a vector with at least one nonzero element) $\mathbf{v} \in \mathbb{C}^n$ such that $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$. The *spectral radius* of a square matrix $\mathbf{A}\ \in \mathbb{R}^{n \times n}$, denoted as $\rho(\mathbf{A})$, is the maximum of absolute values of its eigenvalues (this value always exists).

A matrix $\mathbf{A}\ \in \mathbb{R}^{n \times n}$ is called a *nonnegative matrix* if all elements of $\mathbf{A}$ are greater or equal to zero. A nonnegative matrix $\mathbf{A}\ \in \mathbb{R}^{n \times n}$ having row sums equal to 1 is called a *stochastic matrix*. A nonnegative matrix $\mathbf{A}\ \in \mathbb{R}^{n \times n}$ having row sums less than or equal to 1 with at least one row sum less than 1 is called a *substochastic matrix*.

The *graph* $\mathcal{G}(\mathbf{A})$ of $\mathbf{A} \in \mathbb{R}^{n \times n}$ is defined to be the directed graph on n nodes $\{v_1, \ldots, v_n\}$ such that there is a directed edge leading from v_i to v_j iff $\mathbf{A}[i, j] \neq 0$. We define a relation $\sim$ on the indices of $\mathcal{G}(\mathbf{A})$ such that $v_i \sim v_j$ iff there exists a path from v_i to v_j and from v_j to v_i. The $\sim$ relation is an equivalence relation, which partitions the nodes of $\mathcal{G}(\mathbf{A})$ into equivalence classes called the *access-equivalent classes* of $\mathbf{A}$. For a matrix $\mathbf{A}$ and an access-equivalent class B of $\mathbf{A}$, we use $\mathbf{A}[B]$ to denote a submatrix of $\mathbf{A}$ with row indices and column indices in B. For a nonnegative matrix $\mathbf{A}\ \in \mathbb{R}^{n \times n}$ with a spectral radius ρ, we define a *basic class* of $\mathbf{A}$ as an access-equivalent class B of $\mathbf{A}$ with $\rho(\mathbf{A}[B]) = \rho$.

Theorem 1 ([14])**.** *The set of basic classes of a nonnegative matrix is always nonempty.*

The last notion that we introduce are reducible and irreducible matrices. However, first, we introduce a notion of a permutation matrix. A permutation matrix is a matrix created by a rearrangement of the rows of the identity matrix. A nonnegative matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is then said to be a *reducible matrix* if there exists a permutation matrix $\mathbf{P}$ such that

$$\mathbf{P}^\top \mathbf{A}\mathbf{P} = \begin{pmatrix} \mathbf{X} & \mathbf{Y} \\ \mathbf{0} & \mathbf{Z} \end{pmatrix} \tag{2.1}$$

where $\mathbf{X}$ and $\mathbf{Z}$ are both square matrices and $\mathbf{Y}$ is an arbitrary matrix. An expression $\mathbf{P}^\top \mathbf{A}\mathbf{P}$ is called a symmetric permutation of $\mathbf{A}$. The effect of a symmetric permutation is interchanging of rows and the corresponding columns of $\mathbf{A}$. The intuition behind reducible matrices is that the graph of a reducible matrix contains a node that cannot reach some other node. If a matrix is not reducible, it is said to be an *irreducible matrix*.

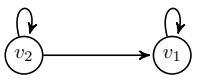
Example 2. *Consider a matrix*

$$\mathbf{A} = \begin{pmatrix} 2 & 0 \\ 3 & 4 \end{pmatrix}. \tag{2.2}$$

Then, this matrix is reducible because there exists a permutation matrix $\mathbf{P} = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ *such that*

$$\mathbf{P}^\top \mathbf{A}\mathbf{P} = \begin{pmatrix} 4 & 3 \\ 0 & 2 \end{pmatrix}. \tag{2.3}$$

Furthermore, the graph $\mathcal{G}(\mathbf{A})$ *looks as follows:*



In this graph, node v_1 *cannot reach node* v_2*.*

The following theorems describe relations between irreducibility and the matrix graph, and restrict the spectral radius of an irreducible substochastic matrix.

Theorem 3 ([14])**.** *A nonnegative matrix* $\mathbf{A} \in \mathbb{R}^{n\times n}$ *is irreducible if and only if* $\mathcal{G}(\mathbf{A})$ *is strongly connected.*

Theorem 4 ([16, 26])**.** *Let* $\mathbf{A} \in \mathbb{R}^{n\times n}$ *be an irreducible substochastic matrix. Then* $\rho(\mathbf{A}) < 1$*.*

The last theorem we introduce describes convergence conditions of matrix powers, which we will later need for a computing the sum of weights of all strings from a language.

Theorem 5 ([14])**.** *Let* $\mathbf{A} \in \mathbb{R}^{n\times n}$ *be a nonnegative matrix. Then, the following are equivalent:*

1. $\mathbf{A}$ *is convergent (i.e.,* $\lim_{n\to\infty} \mathbf{A}^n = \mathbf{0}$*),*
2. $\rho(\mathbf{A}) < 1$*, and*
3. $\mathbf{I} - \mathbf{A}$ *is invertible and* $(\mathbf{I} - \mathbf{A})^{-1} \geq \mathbf{0}$*.*

Further, when the above conditions hold, $\sum_{t=0}^{\infty} \mathbf{A}^t = (\mathbf{I} - \mathbf{A})^{-1}$*.*

2.2 Distance Functions, Distributions, Languages

In this section, we briefly look at distance functions, distributions, formal languages, and also introduce a concept of a probabilistic language distance. This notion will later be used to steer the approximate reduction methods that we propose. The definitions related to formal languages are based on [17].

Distance Functions and Distributions. Given a domain D, let μ be a function $\mu : D \to [0, 1]$. We lift μ to subsets $E \subseteq D$ as follows: $\mu(E) = \sum_{e \in E} \mu(e)$. We call μ a *semi-distribution* if $\mu(D) \leq 1$, and a *probability distribution function* or (*distribution* for short) if $\mu(D) = 1$ [12].

Definition 6. *A* metric *or a* distance function *on a set X is a function $d : X \times X \to \mathbb{R}_{\geq 0}$ such that, for all $x, y, z \in X$, the following conditions are satisfied:*

1. $d(x, y) \geq 0$,
2. $d(x, y) = 0 \iff x = y$,
3. $d(x, y) = d(y, x)$, *and*
4. $d(x, z) \leq d(x, y) + d(y, z)$.

Further, a pseudometric *on X is a function $d : X \times X \to \mathbb{R}_{\geq 0}$ satisfying the axioms for a metric where the second axiom is replaced by the condition $d(x, x) = 0$.*

Formal Languages. An *alphabet* Σ is a finite non-empty set of symbols. A finite sequence of symbols $w = a_1 \dots a_n \in \Sigma^n$, for $n \geq 0$, is called a *word* (or a *string*) over Σ. By $|w| = n$, we denote the *length* of w, and by ε, we denote the *empty word* of length $|\varepsilon| = 0$. Further, we define $\Sigma^0 = \{\varepsilon\}$ and $\Sigma^* = \bigcup_{i \geq 0} \Sigma^i$. Any set $L \subseteq \Sigma^*$ is called a *language* over Σ. Let L_1, L_2 be languages. Then the *concatenation* of L_1 and L_2 is the language $L_1.L_2$ defined as $L_1.L_2 = \{x.y \mid x \in L_1 \wedge y \in L_2\}$.

Now, we define the probabilistic language distance, which describes the distance between two languages according to some distribution on words. This distance expresses the probability that languages L_1 and L_2 over the same alphabet Σ differ on a word from Σ^* chosen randomly according to the given distribution on words.

Definition 7. *Let μ be a distribution over Σ^*. Further, let L_1 and L_2 be languages over Σ. Then, the* probabilistic language distance $d_\mu : 2^{\Sigma^*} \times 2^{\Sigma^*} \to \mathbb{R}_{\geq 0}$ *is defined as*

$$d_\mu(L_1, L_2) = \mu(L_1 \triangle L_2) \tag{2.4}$$

where $A \triangle B$ denotes the symmetric difference of the sets A and B.

For a distribution μ over Σ^*, it can be shown that the function d_μ is a pseudometric [6].

2.3 Finite Automata

In this section, we present a brief introduction into the automata theory, methods of automata reductions, and approaches for comparing their languages. The definitions related to finite automata are taken from [17].

Definition 8. *A* nondeterministic finite automaton *(NFA) over an alphabet* Σ *is a tuple* $A = (Q, \Sigma, \delta, q_0, F)$ *where*

- Q *is a finite non-empty set of states,*
- $\delta : Q \times \Sigma \to 2^Q$ *is a transition function,*
- $q_0 \in Q$ *is an initial state, and*
- $F \subseteq Q$ *is a set of accepting (final) states.*

A finite automaton $A = (Q, \Sigma, \delta, q_0, F)$ *is called* deterministic *(DFA) if* $\forall q \in Q$ *and* $\forall a \in \Sigma$: $|\delta(q, a)| \leq 1$.

In some cases, it is more suitable to use an NFA with multiple initial states $I \subseteq Q$. Therefore, we assume an operation singleInit that converts the input NFA with multiple initial states into an equivalent NFA with just one initial state.

We proceed by defining some notions related to finite automata. A *configuration* of an automaton $A = (Q, \Sigma, \delta, q_0, F)$ is an element from $Q \times \Sigma^*$. The configuration (q_0, x) is called the *initial configuration of* A *on* x. A *step* of A is a relation $\vdash_A \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$, defined as $(q, ax) \vdash_A (p, x)$ iff $p \in \delta(q, a)$. A *computation* is then a finite sequence $C_1, \ldots, C_k$ of configurations such that $C_i \vdash_A C_{i+1}$ for all $i = 1, \ldots, k-1$. An *accepting computation of* A *on* x is a computation $C_0, \ldots, C_m$ of A where C_0 is the initial configuration of A on x and $C_m = (p, \varepsilon)$ for a state $p \in F$. Further, for $q_1, q_2 \in Q$ and $w \in \Sigma^*$, we define an *extended transition function* $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ by $q_2 \in \hat{\delta}(q_1, w) \Leftrightarrow (q_1, w) \vdash_A^* (q_2, \varepsilon)$ where $\vdash_A^*$ is the reflexive and transitive closure of $\vdash_A$.

The *language* $L(A)$ *accepted by* A is defined as

$$L(A) = \{w \in \Sigma^* \mid (q_0, w) \vdash_A^* (p, \varepsilon), p \in F\}, \tag{2.5}$$

and the *class of regular languages* is the class of all languages that are accepted by finite automata. Let $q \in Q$ be a state, then the *language accepted at* q (or the *backward language*) is defined as

$$L_A^{-1}(q) = \{w \in \Sigma^* \mid (q_0, w) \vdash_A^* (q, \varepsilon)\}. \tag{2.6}$$

Definition 9. *An NFA is called* unambiguous *(UFA) if it has at most one accepting computation on every input string.*

For a given NFA $A = (Q, \Sigma, \delta, q_0, F)$ and $S \subseteq Q$, we define the *restriction of* A *to the states in* S as $A_{|S} = (S \cup \{q_0\}, \Sigma, \delta_{|S}, q_0, F \cap S)$. For NFAs A_1, A_2, we denote by $A_1 \cap A_2$ the NFA accepting the language $L(A_1) \cap L(A_2)$, which is constructed using the standard product algorithm [17].

2.3.1 Automata Reductions

The problem of a reduction of finite automata is important in many applications where the maximum automaton size (given by the number of its states) is limited, for example, by technical equipment, or where we care about the efficiency of operations with the automata. The complexity of these operations is often a function of the number of states.

If we consider DFAs, there exists an efficient algorithm for their minimization based on the Myhill-Nerode theorem. Unfortunately, the use of this algorithm for general NFAs requires a prior determinization of the input automaton, which may cause an exponential increase in the number of states (compared to the input NFA). Moreover, the size after

minimization can still be exponentially larger than the size of the input NFA. Therefore, this approach is not practically feasible for NFAs with the large number of states.

An alternative approach is to reduce NFAs directly without determinization. The general NFA state minimization is a PSPACE-hard problem [20], but there still exist some practically feasible algorithms to reduce NFAs. One of those algorithms is based on merging states according to the (maximal) simulation equivalence on states. The definition of the simulation equivalence is based on the notion of the simulation relation [15].

Definition 10 ([15])**.** *A* (forward) simulation *on an NFA* $A = (Q, \Sigma, \delta, q_0, F)$ *is a binary relation* $R \subseteq Q \times Q$ *such that, for any states* $q, r \in Q$ *and* $a \in \Sigma$, $(q, r) \in R$ *holds, only if*

1. $q \in F \implies r \in F$*, and*

2. *for every* $q' \in \delta(q, a)$ *there exists* $r' \in \delta(r, a)$ *such that* $(q', r') \in R$*.*

It can be shown that for each NFA, there exists a unique largest simulation, called the *simulation preorder*. The *simulation equivalence* for a simulation preorder $\preccurlyeq$ is then given as $\preccurlyeq \cap \preccurlyeq^{-1}$. Moreover, there exists a polynomial-time algorithm for computing the largest simulation equivalence [19]. Note that, the dual notion to the forward simulation is the *backward simulation* on an NFA $A = (Q, \Sigma, \delta, q_0, F)$, which is defined as the forward simulation on the NFA $\mathsf{singleInit}(A^{-1})$ where $A^{-1} = (Q, \Sigma, \delta^{-1}, F, \{q_0\})$ and $\forall a \in \Sigma, \forall q, p \in Q : q \in \delta^{-1}(p, a) \Leftrightarrow p \in \delta(q, a)$. Then, the *backward simulation preorder* and the *backward simulation equivalence* are defined in the same way as for the forward simulation.

For the given simulation equivalence, the state merging algorithm merges all states in the same equivalence class. A generalization of the above mentioned simulation reduction is a reduction based on preorders [18]. The main idea behind this approach is computing the simulation preorders $\preccurlyeq_L$ and $\preccurlyeq_R$ for the forward and the backward simulation, respectively. Then, two states p and q can be merged when any of the following conditions is met: **a)** $p \preccurlyeq_R q$ and $q \preccurlyeq_R p$ **b)** $p \preccurlyeq_L q$ and $q \preccurlyeq_L p$ **c)** $p \preccurlyeq_R q$ and $p \preccurlyeq_L q$. However, in contrast with the basic simulation reduction, the preorders $\preccurlyeq_R$ and $\preccurlyeq_L$ must be updated after some two states are merged.

The basic simulation reduction uses the only simulation equivalence (forward or backward) for a reduction. Another approach is to use a composition of the forward and the backward equivalences [3][1]. An automaton is then reduced according to the combined equivalence $\equiv_W$ obtained from the combined preorder $W = \preccurlyeq_R \oplus \preccurlyeq_L^{-1}$ where $\preccurlyeq_R$ and $\preccurlyeq_L$ are the backward and the forward simulation preorders, respectively. The weakening combination operator $\oplus$ is defined as follows: Given two preorders H and S over Q, for $x, y \in Q$, $(x, y) \in H \oplus S$ iff **a)** $(x, y) \in H \circ S$ **b)** $\forall z \in Q : (y, z) \in H \implies (x, z) \in H \circ S$.

The above mentioned methods use the merging of states for a reduction only. This is not, however, the only possible approach. The state merging methods can be combined with, e.g., a removing of transitions [25][2] (the RABIT and Reduce tools).

So far, we dealt with language-preserving reductions only where for an NFA A and the reduced NFA A' it holds that $L(A) = L(A')$. There also exists a way of reducing automata, called *hyperminimization*, which modifies the language of an input automaton. A hyperminimizing algorithm converts the input automaton A into a smaller automaton A' such that the symmetric difference between languages $L(A)$ and $L(A')$ is a finite set [4].

[1]The results in this paper are presented in the context of tree automata. However, the results in [3] can be carried over to the classical finite automata.

[2]The methods in this paper are described in the context of Büchi automata, but they carry over to the case of the classical finite automata.

Hyperminimization, however, is not suitable for our purposes because reduction up to a finite difference need not yield a sufficiently small automaton. Moreover, by this reduction, we might remove important strings from the input language.

2.3.2 Automata Language Comparison

In this section, we discuss methods for measuring the difference of languages of finite automata. One of the ways is based on comparing the similarity of strings from the languages as follows. The similarity of two strings can be expressed as a cost of operations transforming a string of a source language to some string of a target language. Such operations considered in the literature are symbol insertion, deletion, or substitution by a different symbol. Then, the minimal cost of a sequence of these operations transforming a string x into a string y is called the *edit-distance* of strings x and y (denoted as $d(x, y)$) [27]. The notion of edit-distance between strings can be generalized to languages. The edit-distance of two languages L_1 and L_2 over Σ is then defined by

$$d(L_1, L_2) = \inf\{d(x, y) \mid x \in L_1, y \in L_2\}. \tag{2.7}$$

The reason why this definition is not suitable for many applications, including ours, is the fact that if languages L_1 and L_2 have at least one common string, then their edit-distance is zero.

If we limit ourselves to the class of regular languages, there exists a polynomial-time algorithm for computing their edit-distance. On the other hand, if we consider context-free languages, then the problem of determining their edit-distance is undecidable [27].

Another approach to comparing regular languages is using the Jaccard distance and the Cesaro-Jaccard distance [30]. For two regular languages L_1 and L_2 over Σ, the $n_{\leq}$ *Jaccard distance* is defined as

$$J_n(L_1, L_2) = \frac{|W_{\leq n}(L_1 \triangle L_2)|}{|W_{\leq n}(L_1 \cup L_2)|} \tag{2.8}$$

where $W_{\leq n}(L)$ denotes the set of words in L of length at most n. For the case $|W_{\leq n}(L_1 \cup L_2)| = 0$, it is defined $J_n(L_1, L_2) = 0$. Because of the existence of infinite regular languages, the length of their strings cannot be bounded by some fixed n, and therefore, a natural step to quantify the distance of those languages is to take the limit of the Jaccard distance. It can, however, be shown that this limit does not always exist. For this reason, there exists a generalization of the previous Jaccard distance, called the Cesaro-Jaccard distance. For two regular languages L_1 and L_2, the *Cesaro-Jaccard distance* is defined as

$$J_C(L_1, L_2) = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} J_i(L_1, L_2). \tag{2.9}$$

In the paper [30], it is shown that the Cesaro-Jaccard distance is well-defined (i.e., the above limit always exists). The actual computation is, however, quite complicated because of a need to determine the matrix polynomials of the automaton adjacency matrix when computing the Cesaro-Jaccard distance. Moreover, for our purposes, it seems more practical to reflect in the distance the fact that not all the strings are of the same importance for us, e.g., some of them appear rarely or do not appear at all. Hence, an approximate reduction that makes a mistake by classifying such strings should be better than on that makes a mistake on more important strings. Therefore, for a comparison of the languages of automata used in network traffic monitoring we will use the proposed probabilistic distance. That is why, we now aim at weighted automata, which allows us to assign a weight to a string. This weight can be interpreted as the importance of that string.

2.4 Weighted and Probabilistic Finite Automata

This section is devoted to weighted finite automata and their special case—probabilistic finite automata. Many different kinds of probabilistic automata can be defined using different forms of their transition function, e.g., bundle probabilistic automata, Pneuli-Zuck automata, and others [32]. Here, we consider simple probabilistic automata only. We start with the definition of a general weighted automaton over $\mathbb{R}_{\geq 0}$. In this section, we use the formalism based on [5] and basic definitions based on [5, 28].

Definition 11 ([5, 28])**.** *We define a* weighted finite automaton *(WFA) over* Σ *with* n *states as a triple of the form* $A = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ *where* $\boldsymbol{\alpha}_0 \in \mathbb{R}^n_{\geq 0}$ *is a vector of* initial weights*,* $\boldsymbol{\alpha}_f \in \mathbb{R}^n_{\geq 0}$ *is a vector of* final weights*, and, for all* $a \in \Sigma$*,* $\mathbf{A}_a \in \mathbb{R}^{n\times n}_{\geq 0}$ *is the* transition matrix *for symbol* a*. We call the set* $Q_A = \{q_1, \ldots, q_n\}$ *the* states *of* A[3]*. Further, the set* $I_A = \{q_i \in Q_A \mid \boldsymbol{\alpha}_0[q_i] \neq 0\}$ *is called the set of* initial states *and* $F_A = \{q_i \in Q_A \mid \boldsymbol{\alpha}_f[q_i] \neq 0\}$ *the set of* final states*.*

Now, we get to the definition of a (semi-)probabilistic automaton, which is a special case of a general weighted finite automaton.

Definition 12. *We define a* semi-probabilistic automaton *(SPA) over* Σ *as a WFA* $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ *where* $\boldsymbol{\alpha}_0 \in [0,1]^n$ *is called the vector of* initial probability values*,* $\boldsymbol{\alpha}_f \in [0,1]^n$ *is called the vector of* final probability values*, and for all* $a \in \Sigma$*,* $\mathbf{A}_a \in [0,1]^{n\times n}$*. Moreover, the following two conditions are required to hold for* P*:*

1. *The initial probability values of all states sum up to 1, i.e.,*

$$\sum_{i=1}^{n} \boldsymbol{\alpha}_0[i] = 1. \tag{2.10}$$

2. *For every state* q_i *such that* $1 \leq i \leq n$*, the probabilities of all outgoing transitions plus the probability of acceptance sum up to at most 1, i.e.,*

$$\left(\sum_{j=1}^{n}\sum_{a\in\Sigma} \mathbf{A}_a[i,j]\right) + \boldsymbol{\alpha}_f[i] \leq 1. \tag{2.11}$$

Further, we define a probabilistic automaton *(PA) as an SPA where the inequality "*≤ 1*" in 2.11 is substituted with equality "*$= 1$*" [11].*

For P being an SPA over Σ, a *support* of P is an NFA obtained by omitting the weights from P. The support of an SPA P is denoted by $\mathsf{supp}(P)$. For an SPA $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$, the operation $\mathsf{supp}(P)$ is defined as $\mathsf{supp}(P) = \mathsf{singleInit}(A)$ where $A = (Q_P, \Sigma, \delta', I_P, F_P)$, and, for each $p, q \in Q_P$: $q \in \delta'(p, a)$ iff $\mathbf{A}_a[p, q] > 0$.

An SPA (PA) P is called a *deterministic SPA (PA)*, denoted as DSPA (DPA), if P has a single initial state and $\mathsf{supp}(P)$ is a deterministic finite automaton.

Given a word $w = a_1 \ldots a_m \in \Sigma^*$ of length $m \geq 0$ and a WFA P, we define the *weight of* w *in* P (or *probability of* w *in* P in the case of an SPA), denoted $f_P(w)$, as

[3] Without loss of generality, we assume that the transition matrix and the vectors of initial and final weights can be indexed by the states from Q_A, i.e., that there exists some fixed bijection $\varphi : Q_A \to \{1, \ldots, |Q_A|\}$.

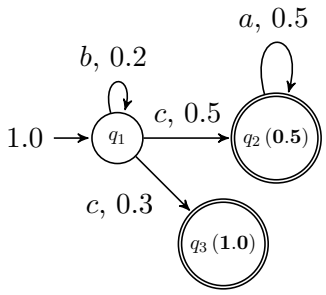


Figure 2.1: Example of PA over alphabet $\{a, b, c\}$. The accepting weight is denoted by a bold number in brackets inside of the accepting state. The initial weight is denoted by a number before the initial states.

$f_P(w) = \boldsymbol{\alpha}_0^\top \cdot \mathbf{A}_{a_1} \cdots \mathbf{A}_{a_m} \cdot \boldsymbol{\alpha}_f$. Further, we denote $\mathbf{A}_{a_1} \cdots \mathbf{A}_{a_m}$ as $\mathbf{A}_w$, and we let $\mathbf{A}_\varepsilon = \mathbf{I}$ and $\mathbf{A}_\Sigma = \sum_{a\in\Sigma} \mathbf{A}_a$. The pointwise extension of f_P to a language L is defined as $f_P(L) = \sum_{w\in L} f_P(w)$. For a WFA P and $q \in Q_P$, the *language accepted at* q is defined as $L_P^{-1}(q) = L_{\mathsf{supp}(P)}^{-1}(q)$.

Example 13. *Consider a PA P over alphabet $\{a, b, c\}$ given in Figure 2.1. Then, the values of the function f_P for strings bcaa, c, and bcc, respectively, are given as*

$$f_P(bcaa) = 1.0 \cdot 0.2 \cdot 0.5 \cdot 0.5 \cdot 0.5 \cdot 0.5 = 0.0125, \tag{2.12}$$

$$f_P(c) = 1.0 \cdot 0.5 \cdot 0.5 + 1.0 \cdot 0.3 \cdot 1.0 = 0.55, \tag{2.13}$$

$$f_P(bcc) = 0. \tag{2.14}$$

Further, $f_P(\{bcaa, c, bcc\}) = 0.5625$. The language accepted at q_2 can be expressed by the regular expression $L_P^{-1}(q_2) = b^*ca^*$.

We say that two WFAs A_1 and A_2 over an alphabet Σ are *equivalent* iff $f_{A_1}(w) = f_{A_2}(w)$ for all $w \in \Sigma^*$.

A state q of a WFA $A = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ is said to be *non-accessible* if there is no path in $\mathcal{G}(\mathbf{A}_\Sigma)$ from any state in I_A to q and *non-coaccessible* if there is no path in $\mathcal{G}(\mathbf{A}_\Sigma)$ from q to any state in F_A. A WFA A is *well-formed* if it has no non-coaccessible states. If A has no non-accessible and no non-coaccessible states, it is called *trimmed*.

The following lemma, which characterizes the function represented by a WFA, was in a slightly different form presented in [5] but without a proof of Equality 2.15, so we state it here in a complete form. An intuition behind this lemma is that we can express the value $f_P(\Sigma^t)$ as a matrix power and using the results from linear algebra we are able to sum these matrices for each t.

Lemma 14. *Let $A = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ be a WFA such that $\rho(\mathbf{A}_\Sigma) < 1$. Then $f_A(\Sigma^*) = \boldsymbol{\alpha}_0^\top (\mathbf{I} - \mathbf{A}_\Sigma)^{-1} \boldsymbol{\alpha}_f$.*

Proof. First, we prove an auxiliary equality

$$\sum_{w\in\Sigma^t} \mathbf{A}_w = \mathbf{A}_\Sigma^t \quad \text{for all } t \in \mathbb{N}. \tag{2.15}$$

By induction on t. For $t = 0$, the equality is valid since $\mathbf{A}_\varepsilon = \mathbf{A}_\Sigma^0 = \mathbf{I}$. Further, we assume that the equality holds for $t = s$. We prove that it also holds for $t = s + 1$:

$$\sum_{w\in\Sigma^{t+1}} \mathbf{A}_w = \sum_{w\in\Sigma^t} \sum_{a\in\Sigma} \mathbf{A}_w \mathbf{A}_a = \left(\sum_{w\in\Sigma^t} \mathbf{A}_w\right) \cdot \left(\sum_{a\in\Sigma} \mathbf{A}_a\right) = \mathbf{A}_\Sigma^t \mathbf{A}_\Sigma = \mathbf{A}_\Sigma^{t+1}. \tag{2.16}$$

Finally, according to Theorem 5 and Equality 2.15, we get

$$f_A(\Sigma^*) \quad = \quad \sum_{t\in\mathbb{N}} \sum_{w\in\Sigma^t} \boldsymbol{\alpha}_0^\top \mathbf{A}_w \boldsymbol{\alpha}_f = \sum_{t\in\mathbb{N}} \boldsymbol{\alpha}_0^\top \mathbf{A}_\Sigma^t \boldsymbol{\alpha}_f = \boldsymbol{\alpha}_0^\top (\mathbf{I} - \mathbf{A}_\Sigma)^{-1} \boldsymbol{\alpha}_f. \tag{2.17}$$

□

2.4.1 Weighted Automata Operations

In this section, we look at weighted automata operations used in the rest of this thesis.

Definition 15 ([28])**.** *Let A_1 and A_2 be WFAs over Σ and $f_{A_1}, f_{A_2} : \Sigma^* \to \mathbb{R}_{\geq 0}$ be functions generated by these automata. Then the* intersection *(or* Hadamard product*) of A_1 and A_2 is the WFA $A_1 \cap A_2$ over Σ generating the function $f_{A_1 \cap A_2}$ that is defined as*

$$\forall x \in \Sigma^* : f_{A_1\cap A_2}(x) = f_{A_1}(x) \cdot f_{A_2}(x). \tag{2.18}$$

There exists an efficient algorithm for computing the intersection of two arbitrary WFAs. Algorithm 1 was taken from [28], and we state it here in the form that reflects our notions and our definition of WFAs.

Algorithm 1: Intersection of two WFAs

Input: WFA $A_1 = (\boldsymbol{\alpha}_0^1, \boldsymbol{\alpha}_f^1, \{\mathbf{A}_a^1\}_{a\in\Sigma})$, $A_2 = (\boldsymbol{\alpha}_0^2, \boldsymbol{\alpha}_f^2, \{\mathbf{A}_a^2\}_{a\in\Sigma})$
Output: WFA $A = A_1 \cap A_2$

```
1:  foreach q = (q1, q2) ∈ Q_{A1} × Q_{A2} do
2:      α0[q] ← α0^1[q1] · α0^2[q2]
3:      αf[q] ← αf^1[q1] · αf^2[q2]
4:      foreach q' = (q1', q2') ∈ Q_{A1} × Q_{A2} do
5:          foreach a ∈ Σ do
6:              A_a[q, q'] ← A_a^1[q1, q1'] · A_a^2[q2, q2']
7:          end
8:      end
9:  end
10: return (α0, αf, {A_a}_{a∈Σ})
```

The states of the WFA $A_1 \cap A_2$ are pairs of the form (q_1, q_2) where $q_1 \in Q_{A_1}$ and $q_2 \in Q_{A_2}$. The presented algorithm does not remove non-accessible and non-coaccessible states. In the rest of this thesis, if we talk about a WFA $A_1 \cap A_2$, we implicitly mean the WFA constructed using Algorithm 1.

In the following text, we consider the operation $\mathsf{weighted}(A)$ that transforms an NFA A into a WFA by adding a unit weight to each edge of A. More precisely, let $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA over Σ and $\varphi : Q \to \{1, \ldots, |Q|\}$ be an arbitrary bijective function. Then, $\mathsf{weighted}(A) = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ where

$$\forall q \in Q : \boldsymbol{\alpha}_0[\varphi(q)] = \begin{cases} 1 & \text{if } q = q_0, \\ 0 & \text{otherwise,} \end{cases} \quad \boldsymbol{\alpha}_f[\varphi(q)] = \begin{cases} 1 & \text{if } q \in F, \\ 0 & \text{otherwise,} \end{cases}$$

$$\forall q_1, q_2 \in Q, \forall a \ \in \Sigma : \mathbf{A}_a[\varphi(q_1), \varphi(q_2)] = \begin{cases} 1 & \text{if } q_2 \in \delta(q_1, a), \\ 0 & \text{otherwise.} \end{cases} \tag{2.19}$$

Definition 16. *Let P be an SPA and A be an NFA. Then, the intersection $\odot$ of P and A is the WFA defined as $P \odot A = P \cap \mathsf{weighted}(A)$.*

Other considered operations over WFAs are trim and $\mathsf{wellFormed}$. The $\mathsf{trim}(A)$ operation converts a WFA A to an equivalent trimmed WFA. Similarly, the $\mathsf{wellFormed}(A)$ operation converts a WFA to an equivalent well-formed WFA. There exists an algorithm for the trim operation (Algorithm 2, modified from [35], based on an algorithm for graph reachability). An algorithm for the operation $\mathsf{wellFormed}$ can be obtained from Algorithm 2 (by simply replacing the line 10 by $Q \leftarrow F_j$).

Algorithm 2: The Trim operation

Input: WFA $A = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$
Output: WFA $\mathsf{trim}(A)$

1: $i \leftarrow 0$, $I_0 \leftarrow I_A$, $j \leftarrow 0$, $F_0 \leftarrow F_A$
2: **repeat**
3: $\quad I_{i+1} \leftarrow I_i \cup \{q \in Q_A \mid \exists p \in I_i \, \exists a \ \in \Sigma : \mathbf{A}_a[p, q] > 0\}$
4: $\quad i \leftarrow i + 1$
5: **until** $I_i = I_{i-1}$;
6: **repeat**
7: $\quad F_{j+1} \leftarrow F_j \cup \{p \in Q_A \mid \exists q \in F_j \, \exists a \ \in \Sigma : \mathbf{A}_a[p, q] > 0\}$
8: $\quad j \leftarrow j + 1$
9: **until** $F_j = F_{j-1}$;
10: $Q \leftarrow I_i \cap F_j$
11: **return** $(\boldsymbol{\alpha}_0[Q], \boldsymbol{\alpha}_f[Q], \{\mathbf{A}_a[Q]\}_{a\in\Sigma})$, where $\boldsymbol{\alpha}[Q]$ is subvector of $\boldsymbol{\alpha}$ with elements indices in Q.

2.4.2 Learning Probabilistic Automata

Although, learning of probabilistic automata is not the topic of this thesis, we use learning to obtain a PA from the captured network traffic. For this reason, we very briefly describe basic methods used for learning of probabilistic automata from a finite multiset of strings S. This section is based on [12, 13, Chapter 16]. One of the well known is the Alergia algorithm, which is based on a merging of states. The Alergia algorithm constructs a probabilistic automaton from an initial frequency prefix tree acceptor.

The *frequency prefix tree acceptor of S* (denoted as $\mathsf{FPTA}(S)$) is a tree-shaped DFA (prefix tree) accepting S [13]. Moreover, each transition is associated with the number of strings from S that traverse this transition and each state is associated with the number of strings that are accepted in this state. The general *frequency finite automaton* (FFA) is then a weighted automaton with the positive integer weights (i.e., the transition matrix is a matrix over $\mathbb{N}$ and the vector of initial and final weights are vectors over $\mathbb{N}$) [13]. Hence, the frequency prefix tree acceptor is a special case of the frequency finite automaton. For a state q of FFA A, a frequency of q, denoted as $\mathsf{Freq}(q)$, is the sum of the final weight of q and the weights of all outgoing transitions leading from q.

Example 17. *Let us consider a multiset of strings*[4] $S = \{\varepsilon(3), aa(5), ab(2), a(4), ba(8)\}$. *The frequency prefix tree acceptor $\mathsf{FPTA}(S)$ corresponding to S is then shown in Figure 2.2.*

[4]The numbers in brackets denote the number of occurrences in S.

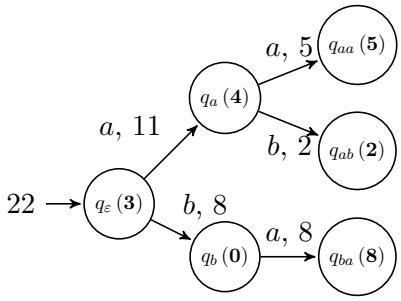


Figure 2.2: A FPTA built from 22 strings.

For merging states, Alergia uses two sets of states of the constructed FFA—the red states and the blue states. Initially, the only red state is the root (the initial state) of the FPTA and the blue states are states reachable from the root state in one step. Alergia then repeatedly selects a blue state q_b such that $\mathsf{Freq}(q_b) \geq t_0$. The parameter t_0 is a threshold on the number of strings in S needed for q_b to be considered for merging. If there is a compatible red state q_r, then q_b is merged with q_r. If there is no compatible red state, then q_b is promoted to the red set. Finally, the blue set is updated in order to contain the states reachable from the red states in one step. Two states are compatible, if their frequencies are sufficiently close (uses the Hoeffding bounds). The compatibility depends also on an input parameter α. Hence, this parameter determines when the two states are merged. Since the algorithm works with a deterministic frequency finite automaton A, in the last step, A is converted to a deterministic probabilistic automaton. The conversion is performed by a normalization of integer weights. Alergia in pseudocode is shown in Algorithm 3 [13, Chapter 16].

Algorithm 3: ALERGIA

Input: A sample S, $\alpha > 0$, $t_0 \in \mathbb{N}$
Output: PA P

$A \leftarrow \mathsf{FPTA}(S)$
Initialize *Red* and *Blue* sets from A
while *Choose* q_b *from Blue set s.t.* $\mathsf{Freq}(q_b) \geq t_0$ **do**
    if $\exists q_r \in Red$: $\mathsf{Compatible}(A, q_r, q_b, \alpha)$ **then**
        $A \leftarrow \mathsf{Merge}(A, q_r, q_b)$
    end
    else
        $Red \leftarrow Red \cup \{q_b\}$
    end
    Update *Blue* set
end
return $P = \mathsf{PA}(A)$

However, Alergia is not the only algorithm based on merging states for learning of probabilistic automata. There are variations of this mentioned algorithm that differ on the compatibility test of two states (DSAI algorithm, MDI algorithm, etc.) [12, 13].

Alergia (and its variations) learns both structure and probabilities of a PA. This approach is definitely not the only way of learning probabilistic automata. There are also algorithms, which learn probabilities to a given structure (e.g., Baum-Welch algorithm) [13, Chapter 17].

Chapter 3

Finite Automata in Network Traffic Filtering

In this chapter, we describe an application of finite automata in a framework for HW accelerated network traffic filtering. This chapter is based on [23, 24, 21, 7]. Due to a massive expansion of computer networks, and especially of the Internet, the communication with other computers all over the world became a quite common thing nowadays. Moreover, this expansion is still accelerating, e.g., due to concepts such as the Internet of Things (IoT). However, this possibility of connection and communication brings also new possibilities for malicious users. For this reason, a great effort to improve the security of networks is being made. The used methods include among others detection of malicious activities such as attacks. One of the technologies that try to detect these malicious activities are network intrusion detection systems (NIDS). The task of an NIDS is among others to analyze packets in order to detect hostile traffic. Examples of such systems are Snort[1] or Bro[2].

The majority of network intrusion detection systems are based on rules that identify malicious traffic. These rules are then applied to each incoming packet. If the input packet is matched against some rule, it can be removed from the traffic (or sent for further analysis). Many used rules are based on *regular expressions* (REs). This means that malicious traffic is described by REs, and an NIDS tries to match the packet payload with at least one RE out of the given set. Regular expressions need not be used only for detection of malicious packets, but also, for instance, for an identification of application protocols of incoming packets (a so-called L7 filter[3]).

3.1 Hardware Acceleration

The main objective of a network traffic filtering system is real-time matching of the traffic against a large set of regular expressions while maintaining a maximum throughput. Common computers do, however, not provide satisfactory computational power for processing large sets of REs against the traffic on multigigabit networks. For this reason, various hardware architectures for RE matching acceleration were constructed. The hardware architectures can then be used for a traffic pre-filtration according to a set of rules (i.e., REs).

[1] `http://www.short.org`
[2] `http://www.bro.org`
[3] `http://l7-filter.sourceforge.net`

Most of these architectures are based on the FPGA[4] technology. The pre-filtration labels suspicious packets, and these packets are further analyzed by the NIDS software. The use of hardware pre-filtration reduces the amount of data processed in software, and therefore, increases the throughput of the whole system. The example of such a system is shown in Figure 3.1.

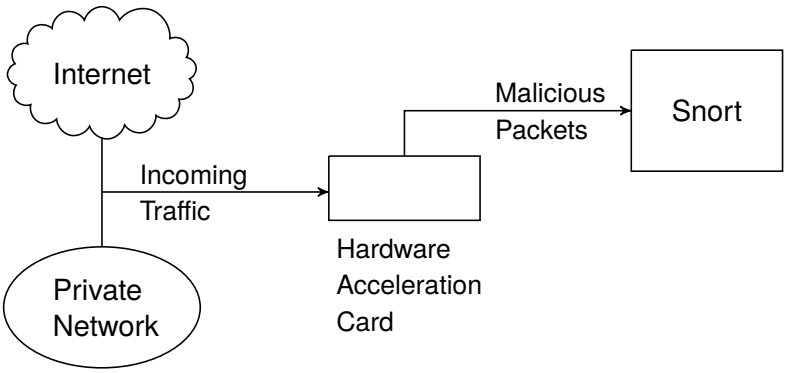


Figure 3.1: The architecture of a network traffic pre-filtration system that uses a hardware acceleration card. The aim of the card is to separate suspicious packets from the input traffic and sent them to a software NIDS (such as Snort or Bro) [23].

The hardware circuit that performs pattern matching is not usually constructed directly from REs. Instead, the REs are first transformed into a finite automaton, and the resulting automaton is then mapped into an FPGA. The main design approaches for network traffic pattern matching in hardware are therefore based on DFAs and NFAs.

An advantage of the DFA approach is that only one state of a deterministic automaton can be active at once, which allows to store the whole transition function into a RAM memory. Hence, it is not necessary to implement the transition function directly as a hardware circuit. This can be appropriate especially in the cases when we need to quickly replace the current automaton with an automaton implementing different REs. However, the use of the DFA approach is limited by the size of the memory available in the HW device. This limitation is made worse by the fact that a DFA representing a given set of rules may be exponentially bigger then an NFA representing the same language.

On the other hand, using NFAs can lead to much smaller automata. The transition function of an NFA, obtained by a conversion from REs, is synthesized directly as a hardware circuit, which is capable of efficiently handling the fact that the NFA can be in multiple states after reading some word. The NFA-based approach is of course limited by the size and the capacity of the FPGA chip. That is the reason why many methods and techniques for minimization of the consumed FPGA resources were introduced [7]. One of these methods is, for example, a use of multi-character decoders, which are able to process more characters per clock cycle. An example of a circuit representing an NFA is shown in Figure 3.2. A disadvantage of the NFA approach can be the long time it takes to change the NFA synthesized in hardware.

[4]Field Programmable Gate Array

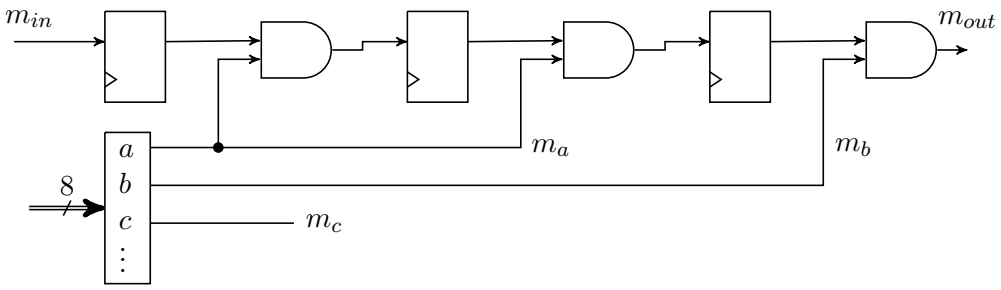


Figure 3.2: An example of the NFA-based circuit realization of the language $L = \{aab\}$ [7]. The circuit consists of a character decoder, flip-flops, and AND gates. In this example, each flip-flop represents a state of the NFA accepting the language L (if a flip-flop is set, then this corresponding state of the NFA is active). By the signal m_{in}, the first flip-flop is set (it represents an initial state of the NFA). In each cycle, next 8-bit character σ from the input is decoded, and according to the result, the signal m_σ is set. The AND gates, based on the character signals, steer the setting of flip-flops (i.e., changing of the active states of the NFA). The output signal m_{out} then signalizes whether the input string is accepted.

3.2 Network Automata Reductions

Since the REs are translated first into NFAs, possible determinized, and then mapped to FPGAs, it is desirable to reduce the size of the involved automata (either DFAs or NFAs) as much as possible. The reduction can save resources of the target hardware architecture such as the size of the circuit. The used techniques for the automata size reduction depend on the type of the automaton. If we require a use of deterministic automata, we can convert these NFAs into minimal DFAs. If we are able to operate with nondeterministic automata, we can use the various NFA reductions discussed in Section 2.3.1, and/or the reductions that we propose in this work in Chapter 5. The workflow of this process with the reduction is shown in Figure 3.3.

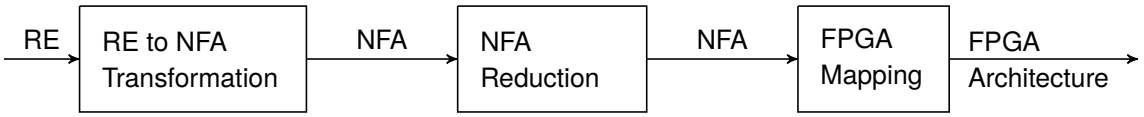


Figure 3.3: The NFA reduction in a network traffic filtering [24].

In some cases, we need the information about which RE is matched. For example, some applications can identify the application protocol of incoming packets according to the matched REs. For these applications, the standard simulation reduction is not usable because standard nondeterministic finite automata do not distinguish final states and the simulation reduction can merge final states. For this reason, the concept of the multi-language nondeterministic finite automaton (MNFA) was introduced [24]. The only difference between NFA and MNFA is that MNFA defines a set of labels and every final state is marked by some label. The simulation equivalence on MNFAs can be defined in a very similar way as for NFAs, one just has to ensure that the final states with different labels are not in the equivalence relation [24].

Chapter 4

Computing Probabilistic Distance for Regular Languages

In this chapter, we propose our algorithm for a computation of the probabilistic distance of regular languages. We show that if the languages are given by UFAs, their probabilistic distance can be computed in polynomial time. In this case, no prior determinization is necessary. For the case when the languages are given by general NFAs, one can use disambiguation to obtain unambiguous automata. This approach can be more efficient compared to an *a priori* NFA determinization.

As we will show later in more detail, the probabilistic distance can be quite useful in the context of HW accelerated network filtering. In particular, a PA can be used to model network traffic (more concretely, occurrences of packets in the network traffic). Then, the probabilistic distance can be used for comparing automata obtained by language non-preserving reductions and for steering such reductions (e.g., by a constraint on the maximal probabilistic distance between an original and the reduced automaton). The workflow of our approach for the automata languages comparison and obtaining a PA in network traffic monitoring is shown in Figure 4.1.

4.1 Probabilistic Distance and SPAs

In this section, we give a lemma useful for the computation of the probabilistic distance. Recall that, for a distribution μ over Σ^* and languages $L_1, L_2 \subseteq \Sigma^*$, the probabilistic distance is defined as $\mu(L_1 \triangle L_2)$. Hence, a crucial role is played by the symmetric difference. Moreover, a straightforward computation of the symmetric difference can be very expensive. That is the case when the languages are given by general NFAs whose symmetric difference is usually computed as $L_1 \triangle L_2 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$. However, then, determinization is needed to compute complements of the languages L_1 and L_2. Fortunately, the below lemma shows that for computing the probabilistic distance it is not necessary to compute the symmetric difference of the input languages directly (and therefore, we avoid of explicit determinization).

Lemma 18. *Let $L_1, L_2 \subseteq \Sigma^*$ and μ be a semi-distribution over Σ^*. Then*

$$\mu(L_1 \triangle L_2) = \mu(L_1) + \mu(L_2) - 2 \cdot \mu(L_1 \cap L_2). \tag{4.1}$$

Proof. We start with the definition of the symmetric difference

$$L_1 \triangle L_2 \quad = \quad (L_1 \setminus L_2) \cup (L_2 \setminus L_1). \tag{4.2}$$

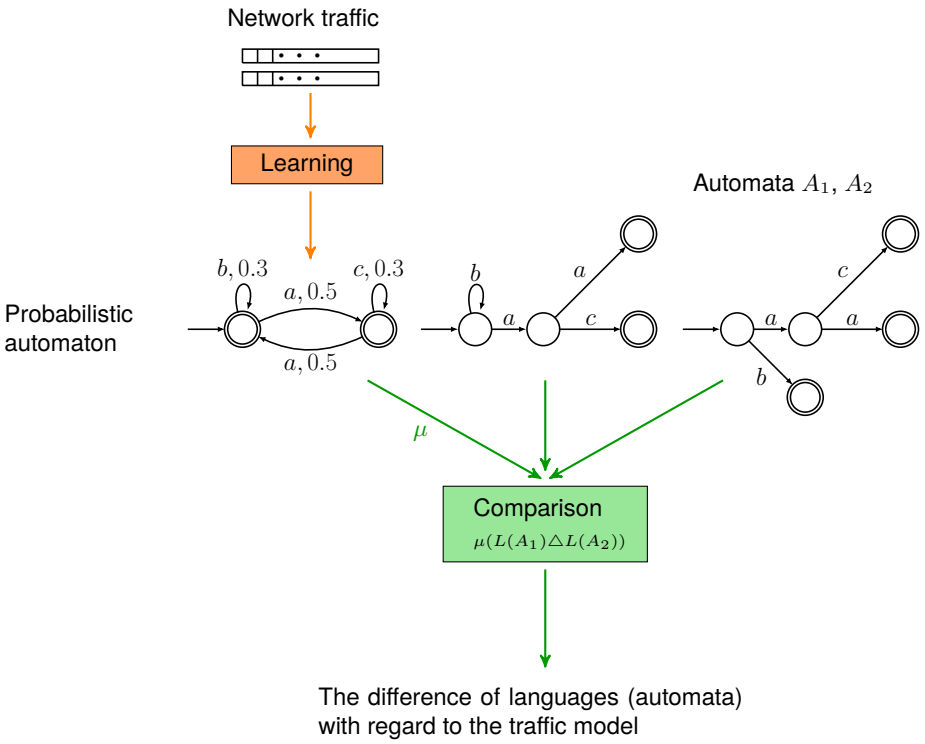


Figure 4.1: The workflow of the language comparison approach that we propose for the context of HW accelerated network traffic filtering. In the first step, a probabilistic automaton is obtained from the input traffic by learning. In the second step, this PA and two NFAs encoding the REs of interest are input for the comparison.

Since $L_1 \setminus L_2$ and $L_2 \setminus L_1$ are disjoint and from the fact that $A \setminus B = A \setminus (A \cap B)$, we get

$$\begin{aligned} \mu((L_1 \setminus L_2) \cup (L_2 \setminus L_1)) &= \sum_{w \in L_1 \setminus L_2} \mu(w) + \sum_{w \in L_2 \setminus L_1} \mu(w) = \\ &= \sum_{w \in L_1 \setminus (L_1 \cap L_2)} \mu(w) + \sum_{w \in L_2 \setminus (L_1 \cap L_2)} \mu(w). \end{aligned} \tag{4.3}$$

Further, from the definition of μ and the fact that $L_1 \cap L_2 \subseteq L_1$, we obtain

$$\sum_{w \in L_1 \setminus (L_1 \cap L_2)} \mu(w) = \sum_{w \in L_1} \mu(w) - \sum_{w \in L_1 \cap L_2} \mu(w) = \mu(L_1) - \mu(L_1 \cap L_2). \tag{4.4}$$

Finally, from Equations 4.3 and 4.4, we have

$$\mu(L_1 \triangle L_2) = \mu(L_1) + \mu(L_2) - 2 \cdot \mu(L_1 \cap L_2). \tag{4.5}$$

□

Due to Lemma 18, we can compute the probabilistic distance by taking $\mu(L_1 \triangle L_2) = \mu(L_1) + \mu(L_2) - 2 \cdot \mu(L_1 \cap L_2)$. For computing the distance we thus need to compute $\mu(L)$ for a distribution μ given by a PA, which is the aim of the following text. The first preliminary step for computing $\mu(L)$ is a computation of $f_P(\Sigma^*)$ for an SPA P. For this, we can apply Lemma 14 but first, we have to show that the condition concerning spectral

radius is satisfied, which is the focus of the below theorem. The proof of this theorem is given in Appendix A.

Theorem 19. *If an SPA* $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ *is well-formed, then* $\rho(\mathbf{A}_\Sigma) < 1$.

Given a well-formed SPA $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$, Theorem 19 allows us to compute $f_P(\Sigma^*)$ as $\boldsymbol{\alpha}_0^\top \cdot (\mathbf{I} - \mathbf{A}_\Sigma)^{-1} \cdot \boldsymbol{\alpha}_f$ (see Lemma 14). Another similar way of computing $f_P(\Sigma^*)$ can be found in [11]. The method in [11] uses a difficult linear algebraic approach (including, e.g., decomposition or projection on subspaces). Our way, on the other hand, uses easier to grasp notions of well-formedness (which has an easy graph representation) together with automata matrix convergence. Therefore, we will further consider only the first mentioned way.

4.2 Computing Probabilistic Distance for UFAs

In this section, we build on the results from the previous section, and we present a polynomial-time algorithm for computing the probabilistic distance $\mu(L_1 \triangle L_2)$ of languages L_1, L_2 given by UFAs. Recall that, the basic idea how to compute the distance was presented in Lemma 18 that expressed the distance as $\mu(L_1 \triangle L_2) = \mu(L_1) + \mu(L_2) - 2 \cdot \mu(L_1 \cap L_2)$. Moreover, we made a first preliminary step for the computation of $\mu(L)$ by the introduction of the way of computing $f_P(\Sigma^*)$ for an SPA P. Now, we will continue with these considerations and we show how to compute $\mu(L)$ when the semi-distribution μ is given by a PA and the regular language L is given by a UFA.

The following two lemmas give us a possible way how to compute $\mu(L)$ using intersection of automata.

Lemma 20. *Let* A *be a UFA and* $f : \Sigma^* \to \mathbb{R}$ *be a function generated by the WFA* $\mathsf{weighted}(A)$. *Then*

$$\forall w \in \Sigma^* : f(w) = \begin{cases} 1 & \text{if } w \in L(A), \\ 0 & \text{otherwise.} \end{cases} \tag{4.6}$$

Proof. Since A is a UFA, there exists at most one accepting run on any word $w \in \Sigma^*$. From the definition of the $\mathsf{weighted}$ operation, we get $f(w) = 1$ for $w \in L(A)$ and $f(w) = 0$ for $w \notin L(A)$. □

Lemma 21. *Let* P *be an SPA over* Σ, A *be a UFA over* Σ, *and* P' *be a WFA such that* $P' = P \odot A$. *Then* $f_P(L(A)) = f_{P'}(\Sigma^*)$.

Proof. We start from the definition of automata intersection, thus $P' = P \cap \mathsf{weighted}(A)$. Further, from Lemma 20, we get $f_P(w) = f_{P'}(w)$ for $w \in L(A)$, and $f_{P'}(w) = 0$ for $w \notin L(A)$. And therefore, $f_P(L(A)) = f_{P'}(\Sigma^*)$. □

Before we show how to compute $\mu(L(A))$ for a UFA A where μ is a distribution given by a PA P, we turn our attention to a simpler case, i.e., the computation of $\mu(L(A))$ (or equivalently written $f_P(L(A))$) for a DFA A. In the first step, we obtain the automaton $P' = \mathsf{wellForm}(P \odot A)$. Since P' is an SPA, in the second step, we compute the value $f_{P'}(\Sigma^*)$ (see the previous section) and according to Lemma 21 we have that $f_{P'}(\Sigma^*) = f_P(L(A))$.

If we want to compute $\mu(L(A))$ for a UFA A, we cannot use exactly this procedure because $P \odot A$ need not be an SPA (for some state the sum of the final weight and the weights of outgoing transitions from this state can be greater than 1). Therefore, we

cannot use Theorem 19 for computing $f_{P'}(\Sigma^*)$, however, we would like to use the same procedure as for DFAs. For this reason, we give the following theorem, which is an analogy of Theorem 19.

Theorem 22. *Let P be a PA, A be a UFA, and P' be a WFA $P' = \mathsf{trim}(P \odot A) = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$. Then $\rho(\mathbf{A}_\Sigma) < 1$.*

The proof of this theorem can be found in Appendix A. Next, in Algorithm 4, we finally introduce the way of computing the probabilistic distance for UFAs. The algorithm is based on Lemma 18 and Theorem 22. For a better insight, see the proof of the following theorem. In addition, below the theorem, we present an example illustrating the construction. In the following text, when we talk about PAs over Σ, we mean trimmed PAs generating distributions over Σ^*. Moreover, we use d_P as an abbreviation for d_{f_P} where P is a PA.

Algorithm 4: Computing probabilistic distance for UFAs

Input: A PA P over Σ generating the distribution μ_P, UFAs A_1 and A_2 over Σ.
Output: $d_{\mu_P}(L(A_1), L(A_2))$

1: $P_1 \leftarrow \mathsf{trim}(P \odot A_1) = (\boldsymbol{\alpha}_0^1, \boldsymbol{\alpha}_f^1, \{\mathbf{A}_a^1\}_{a\in\Sigma})$
2: $P_2 \leftarrow \mathsf{trim}(P \odot A_2) = (\boldsymbol{\alpha}_0^2, \boldsymbol{\alpha}_f^2, \{\mathbf{A}_a^2\}_{a\in\Sigma})$
3: $P_3 \leftarrow \mathsf{trim}(P \odot (A_1 \cap A_2)) = (\boldsymbol{\alpha}_0^3, \boldsymbol{\alpha}_f^3, \{\mathbf{A}_a^3\}_{a\in\Sigma})$
4: **return** $\boldsymbol{\alpha}_0^{1\top}(\mathbf{I} - \mathbf{A}_\Sigma^1)^{-1}\boldsymbol{\alpha}_f^1 +$
5: $\quad \boldsymbol{\alpha}_0^{2\top}(\mathbf{I} - \mathbf{A}_\Sigma^2)^{-1}\boldsymbol{\alpha}_f^2 \; -$
6: $\quad 2\boldsymbol{\alpha}_0^{3\top}(\mathbf{I} - \mathbf{A}_\Sigma^3)^{-1}\boldsymbol{\alpha}_f^3.$

Theorem 23. *Algorithm 4 is correct.*

Proof. First, from Lemma 18, we obtain

$$\begin{aligned} d_{\mu_P}(L(A_1), L(A_2)) &= \mu_P(L(A_1) \triangle L(A_2)) = \\ &= \mu_P(L(A_1)) + \mu_P(L(A_2)) - 2 \cdot \mu_P(L(A_1) \cap L(A_2)) = \\ &= \mu_P(L(A_1)) + \mu_P(L(A_2)) - 2 \cdot \mu_P(L(A_1 \cap A_2)). \end{aligned} \tag{4.7}$$

Further, according to Lemma 21 and the fact that the trim operation does not change the function generated by a WFA, we have

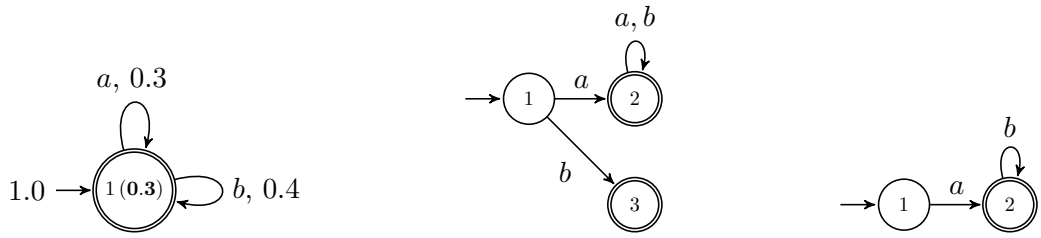
$$\mu_P(L(A_i)) = f_{P_i}(\Sigma^*) \quad \text{for } i \in \{1, 2\}. \tag{4.8}$$

Since the language $L(A_1) \cap L(A_2)$ for UFAs A_1 and A_2 can be easily represented by the UFA $A_1 \cap A_2$, we get

$$\mu_P(L(A_1 \cap A_2)) = f_{P_3}(\Sigma^*). \tag{4.9}$$

Moreover, directly from Theorem 22, we get $\rho(\mathbf{A}_\Sigma^i) < 1$ for $i \in \{1, 2, 3\}$. Finally, using Lemma 14 and Equations 4.7, 4.8, and 4.9, we get the expression on the 4th line of Algorithm 4. □

Before we move to computing the probabilistic distance for general NFAs, we give a short example illustrating the computation of the probabilistic distance of UFAs using the described algorithm. It is shown as Example 24.



(a) A probabilistic automaton P. **(b)** UFA A_1. **(c)** UFA A_2.

Figure 4.2: Examples of input automata over the alphabet $\{a, b\}$ illustrating the computation of probabilistic distance.

Example 24. *Consider the simple PA and the NFAs over an alphabet $\{a, b\}$ given in Figure 4.2. The transition matrices and the vectors of initial and final weights for product WFAs P_i where $i \in \{1, 2, 3\}$ used in Algorithm 4 are the following:*

$$\begin{aligned} \boldsymbol{\alpha}_0^1 &= \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \boldsymbol{\alpha}_f^1 = \begin{pmatrix} 0 \\ 0.3 \\ 0.3 \end{pmatrix}, \quad \mathbf{A}_\Sigma^1 = \begin{pmatrix} 0 & 0.3 & 0.4 \\ 0 & 0.7 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \\ \boldsymbol{\alpha}_0^2 &= \boldsymbol{\alpha}_0^3 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \boldsymbol{\alpha}_f^2 = \boldsymbol{\alpha}_f^3 = \begin{pmatrix} 0 \\ 0.3 \end{pmatrix}, \quad \mathbf{A}_\Sigma^2 = \mathbf{A}_\Sigma^3 = \begin{pmatrix} 0 & 0.3 \\ 0 & 0.4 \end{pmatrix}. \end{aligned} \tag{4.10}$$

The inverse matrices of $\mathbf{I} - \mathbf{A}_\Sigma^i$ are then

$$(\mathbf{I} - \mathbf{A}_\Sigma^1)^{-1} = \begin{pmatrix} 1 & 1 & \frac{2}{5} \\ 0 & \frac{10}{3} & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (\mathbf{I} - \mathbf{A}_\Sigma^2)^{-1} = \begin{pmatrix} 1 & \frac{1}{2} \\ 0 & \frac{5}{3} \end{pmatrix}, \quad (\mathbf{I} - \mathbf{A}_\Sigma^3)^{-1} = \begin{pmatrix} 0 & \frac{1}{2} \\ 0 & \frac{5}{3} \end{pmatrix}. \tag{4.11}$$

Next, the partial sums for computing the probabilistic distance $d_P(L(A_1), L(A_2))$ are given by the following expressions

$$s_1 = \boldsymbol{\alpha}_0^{1\top}(\mathbf{I} - \mathbf{A}_\Sigma^1)^{-1}\boldsymbol{\alpha}_f^1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}^\top \cdot \begin{pmatrix} 1 & 1 & \frac{2}{5} \\ 0 & \frac{10}{3} & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0.3 \\ 0.3 \end{pmatrix} = 0.42, \tag{4.12}$$

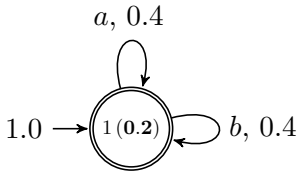
$$s_2 = s_3 = \boldsymbol{\alpha}_0^{2\top}(\mathbf{I} - \mathbf{A}_\Sigma^2)^{-1}\boldsymbol{\alpha}_f^2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}^\top \cdot \begin{pmatrix} 1 & \frac{1}{2} \\ 0 & \frac{5}{3} \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0.3 \end{pmatrix} = 0.15. \tag{4.13}$$

The probabilistic distance of languages $L(A_1)$ and $L(A_2)$ is finally given as

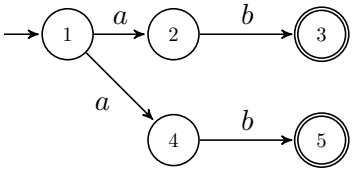
$$d_P(L(A_1), L(A_2)) = s_1 + s_2 - 2s_3 = 0.27. \tag{4.14}$$

4.3 Computing Probabilistic Distance for General NFAs

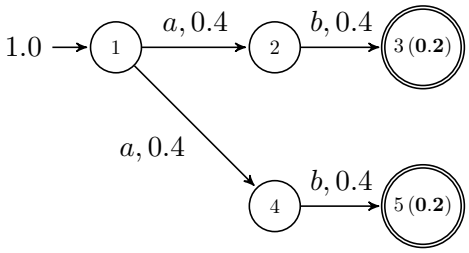
In this section, we investigate the case when the input languages are given by general NFAs. First, however, we show why we cannot use the results from the previous section. Consider the NFA, PA, and their product from Figure 4.3. Then, the value we wish to obtain is $f_P(L(A)) = 1 \cdot 0.4 \cdot 0.4 \cdot 0.2 = 0.032$, but $f_{P'}(\Sigma^*) = 1 \cdot 0.4 \cdot 0.4 \cdot 0.2 + 1 \cdot 0.4 \cdot 0.4 \cdot 0.2 = 0.064$, and therefore $f_P(L(A)) \neq f_{P'}(\Sigma^*)$. The $\odot$-intersection with an ambiguous finite automaton



(a) A probabilistic automaton P over $\{a, b\}$.



(b) A general NFA A over $\{a, b\}$ that is not a UFA.



(c) WFA $P' = P \odot A$.

Figure 4.3: An example of $\odot$-intersection of a PA P and an ambiguous NFA A.

can cause a problem because the probability of a string w in P is multiplied by the number of accepting computations of A on w.

One way how to deal with the mentioned problem is transforming the general NFAs on the input into the equivalent UFAs. The process of transforming an NFA into the equivalent UFA is called disambiguation.

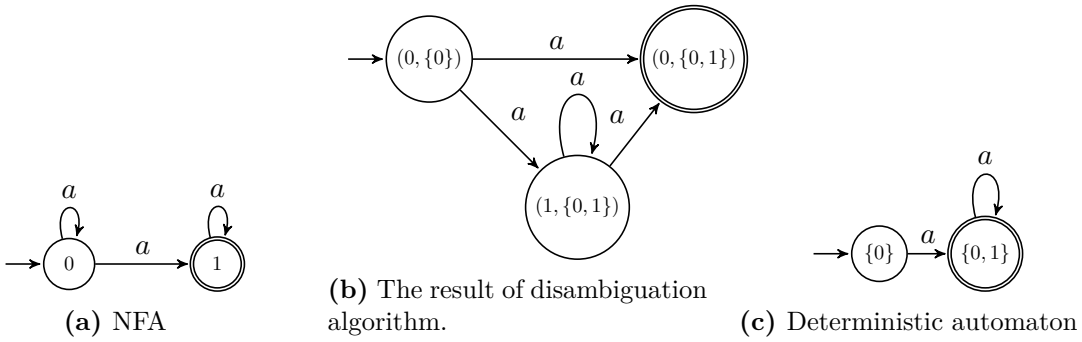
4.3.1 Disambiguation

In this section, we give a modification of an algorithm from [29] for NFA disambiguation. The modification consists of the transformation of the algorithm into our formalism and the correction of typos. Although, determinization is a special case of disambiguation, we use these notions separately. In general, UFAs can be exponentially smaller than deterministic automata. On the other hand, NFAs can be exponentially smaller than UFAs [8]. The presented algorithm from [29] does not require full determinization. There are cases when determinization yields an automaton with exponentially larger number of states (compared to an input automaton), but the disambiguation algorithm creates an automaton with only $\mathcal{O}(n)$ states (where n is the number of states of the input automaton).

Before we move to the algorithm, there remains a question how to decide whether an input NFA is a UFA. The following theorem gives us an answer to this question. Recall that, $A \cap A$ denotes the NFA constructed using the standard product algorithm with the set of states $Q \times Q$ where Q is the set of states of the NFA A.

Theorem 25 ([29])**.** *Let A be a trimmed finite automaton. A is unambiguous iff no state in the automaton* $\mathsf{trim}(A \cap A)$ *is of the form (p, q) with $p \neq q$.*

The algorithm assumes NFAs with a set of initial states. Therefore, we admit a set of initial states for a while. Every NFA with a set of initial states can be converted to an NFA with a single initial state (the singleInit operation). For simplicity, we also use denotation $(p, a, q) \in \delta$ instead of $q \in \delta(p, a)$. Here, we thus assume that δ is a relation on $Q \times \Sigma \times Q$. The algorithm in pseudocode is shown in Algorithm 5.



(a) NFA

(b) The result of disambiguation algorithm.

(c) Deterministic automaton

Figure 4.4: An example of a result of the disambiguation algorithm. Consider the NFA, which is not a UFA, from sub-figure (a). Then the result of the disambiguation algorithm is shown in sub-figure (b). The minimum deterministic variant of the input NFA is shown in sub-figure (c). Note that, in this case, the unambiguous automaton is larger than the deterministic one. In general, however, UFAs can be exponentially smaller than the corresponding minimum DFAs.

The algorithm converts an input NFA $A = (Q, \Sigma, \delta, I, F)$ to an equivalent UFA $A' = (Q', \Sigma, \delta', I', F')$, which is further transformed into a UFA with a single initial state. The set of states Q' contains pairs (p, S) where $p \in Q$ and $S \subseteq Q$. A semantics of a generated state $(p, S) \in Q'$ is the following: If this state (p, S) is reachable by a string w, then for each $s \in S$, w reaches p, s and moreover, there is a string w' such that w' is accepted from both p, s. The algorithm uses a relation $R \subseteq Q' \times Q'$ such that two states of Q' are in R iff both of them can be reached with the same string (from the initial states). All initial states I' are reachable by ε, therefore, they are in R. In the algorithm, the set of states of the original automaton A that can be reached from the states in S by a computation on x is denoted by $\mathsf{reach}(S, x)$.

The queue $\mathcal{Q}$ contains states of A' for future examination. The condition on line 13 ensures that no two final states are reached by the same string. On line 17, we compute the set T of successors of S for a state $(p, S) \in Q'$ and for each transition $(p, a, q) \in \delta$. The set T contains all states r reached from $s \in S$ by reading symbol a such that it also holds that $(r, q) \in Q_B$, which means that r and q can reach a final state with the same string. A new transition $((p, S), a, (q, T))$ is added only if δ' contains no transition from (p', S') to (q, T) labeled by the symbol a where (p', S') is reached by the same string as (p, S) (i.e., states (p', S') and (p, S) are in the relation R). Due to this condition, only one computation from I' to (q, T) on an arbitrary string exists. An example of disambiguation using this algorithm is shown in Figure 4.4.

4.4 Complexity

The overall worst-case complexity of the probabilistic distance computation depends on a type of the input automata. For further considerations, we assume that the alphabet Σ has some fixed size. We start with the case when the input automata are unambiguous.

Unambiguous automata. If the input automata are unambiguous, we can straightforwardly perform Algorithm 4. The product and the following trimming of a PA P and UFA A_1 can be, in the worst case, computed in time $\mathcal{O}(|Q_P|^2 \cdot |Q_{A_1}|^2)$ (the product automaton

has at most $|Q_P| \cdot |Q_{A_1}|$ states and $|\Sigma| \cdot |Q_P|^2 \cdot |Q_{A_1}|^2$ transitions, which gives the mentioned complexity). The transition matrix corresponding to the WFA P_1 has a dimension at most $|Q_P| \cdot |Q_{A_1}|$. Therefore, the inverse matrix can be computed in time $\mathcal{O}((|Q_P| \cdot |Q_{A_1}|)^c)$ where the concrete value c depends on the selected algorithm for matrix inversion. In the case of the Gauss-Jordan elimination, we get $c = 3$ [10]. The algorithms for matrix multiplication and matrix inversion are closely related, and the complexity of matrix inversion depends on the selected algorithm for matrix multiplication[1]. For example, in the case of the Coppersmith-Winograd algorithm, we get $c = 2.376$ [22]. However, we consider only the simplest algorithm, hence the complexity of this step is $\mathcal{O}((|Q_P| \cdot |Q_{A_1}|)^3)$.

Similarly, a dimension of the transition matrix that corresponds to the WFA P_3 is at most $|Q_P| \cdot |Q_{A_1}| \cdot |Q_{A_2}|$. The inverse matrix $(\mathbf{I} - \mathbf{A}_\Sigma^3)^{-1}$ can thus be computed in time $\mathcal{O}((|Q_P| \cdot |Q_{A_1}| \cdot |Q_{A_2}|)^3)$. The overall time complexity of Algorithm 4 is thus given as

$$\mathcal{O}((|Q_P| \cdot |Q_{A_1}|)^3 + (|Q_P| \cdot |Q_{A_2}|)^3 + (|Q_P| \cdot |Q_{A_1}| \cdot |Q_{A_2}|)^3) = \mathcal{O}((|Q_P| \cdot |Q_{A_1}| \cdot |Q_{A_2}|)^3). \quad (4.15)$$

Hence, if the input automata are given as UFAs, the probabilistic distance of their languages can be computed in polynomial time.

General NFAs. If the input automata are ambiguous, we must perform disambiguation (or, alternatively, determinization), which may yield automata with an exponential number of states compared to the input NFA. Hence, if the input automata are given as general NFAs, the probabilistic distance of their languages can be computed in exponential time.

[1] It can be shown that if $M(n)$ denotes the time to multiply two $n \times n$ matrices, then a nonsingular $n \times n$ can be inverted in time $\mathcal{O}(M(n))$ [10].

Algorithm 5: DISAMBIGUATION ALGORITHM

Input: NFA $A = (Q, \Sigma, \delta, I, F)$ with a set of initial states
Output: UFA A' such that $L(A') = L(A)$

$\mathcal{Q} \leftarrow \emptyset$, $Q' \leftarrow \emptyset$, $I' \leftarrow \emptyset$, $\delta' \leftarrow \emptyset$, $F' \leftarrow \emptyset$
$B \leftarrow \mathsf{trim}(A \cap A) = (Q_B, \Sigma, \delta_B, I_B, F_B)$
foreach $i \in I$ **do**
    $S \leftarrow \{i' \mid i' \in I \ \wedge (i, i') \in Q_B\}$
    $Q' \leftarrow Q' \cup \{(i, S)\}$
    $I' \leftarrow Q'$
    $\mathsf{Enqueue}(\mathcal{Q}, (i, S))$
end
$R \leftarrow I' \times I'$
while $\mathcal{Q} \neq \emptyset$ **do**
    $(p, S) \leftarrow \mathsf{Head}(\mathcal{Q})$
    $\mathsf{Dequeue}(\mathcal{Q})$
    if $p \in F$ **and** $\nexists (p', S') \in F'$ *with* $(p', S')R(p, S)$ **then**
        $F' \leftarrow F' \cup \{(p, S)\}$
    end
    foreach $(p, a, q) \in \delta$ **do**
        $T \leftarrow \{r \in \mathsf{reach}(S, a) \mid (q, r) \in Q_B\}$
        if $\nexists ((p', S'), a, (q, T)) \in \delta'$ *with* $(p', S')R(p, S)$ **then**
            if $(q, T) \notin Q'$ **then**
                $Q' \leftarrow Q' \cup \{(q, T)\}$
                $\mathsf{Enqueue}(\mathcal{Q}, (q, T))$
            end
            $\delta' \leftarrow \delta' \cup \{((p, S), a, (q, T))\}$
            foreach (p', S') *such that* $(p', S')R(p, S)$ **and** $((p', S'), a, (q', T')) \in \delta'$ **do**
                $R \leftarrow R \cup \{((q, T), (q', T')), ((q', T'), (q, T))\}$
            end
        end
    end
end
return $\mathsf{singleInit}(A')$ where $A' = (Q', \Sigma, \delta', I', F')$

Chapter 5

Approximate Reduction of Automata

In this chapter, we focus on the approximate reduction of automata in the context of network traffic filtering. We introduce two approaches for the approximate reduction of automata. Since the approximate reduction need not preserve the language of an input NFA, we have to measure the difference between the language of the input NFA and the language of the reduced NFA. For the expression of the difference, we use the probabilistic distance discussed in Chapter 4. Since we want to utilize information from the input network traffic, we again use a probabilistic automaton representing the input traffic for the reduction.

The first proposed approach for the reduction of automata is based on removing branches of the input NFA—we call it the *pruning reduction*. Based on the input probabilistic automaton, the pruning reduction selects branches to be removed from the NFA to be reduced. The second approach is based on adding self-loops—we call it the *self-loop reduction*. The self-loop reduction selects states where a self-loop over every symbol is to be added, followed by removing the redundant states.

The reduction methods are proposed directly for reducing NFAs, however, if the input automaton is unambiguous, more efficient methods for implementing the reduction can be used. The workflow of the automata reductions, including a derivation of the probabilistic automaton to be used, is shown in Figure 5.1. Note that, we implicitly assume that the input automata are trimmed.

5.1 Pruning Reduction

We start with the pruning reduction. As we have already mentioned, the pruning reduction selects branches of the input NFA that are later removed. The choice of branches depends on the input probabilistic automaton. Because this reduction approach is language non-preserving, it is necessary to restrict the reduction by a parameter. Since the reduction removes states, it performs language under-approximation. According to the meaning of the parameter that controls the reduction, we divide the pruning reduction to the *ε-pruning reduction* and the *k-pruning reduction*. In case of the ε-pruning reduction, the parameter sets the maximal error of the reduction. The error is expressed as the probabilistic distance of the input NFA and the reduced NFA. In the case of the k-pruning reduction, the parameter restricts the ratio between the number of states of the reduced NFA and the number of states of the original NFA. An illustration of the pruning reduction is shown in Figure 5.2.

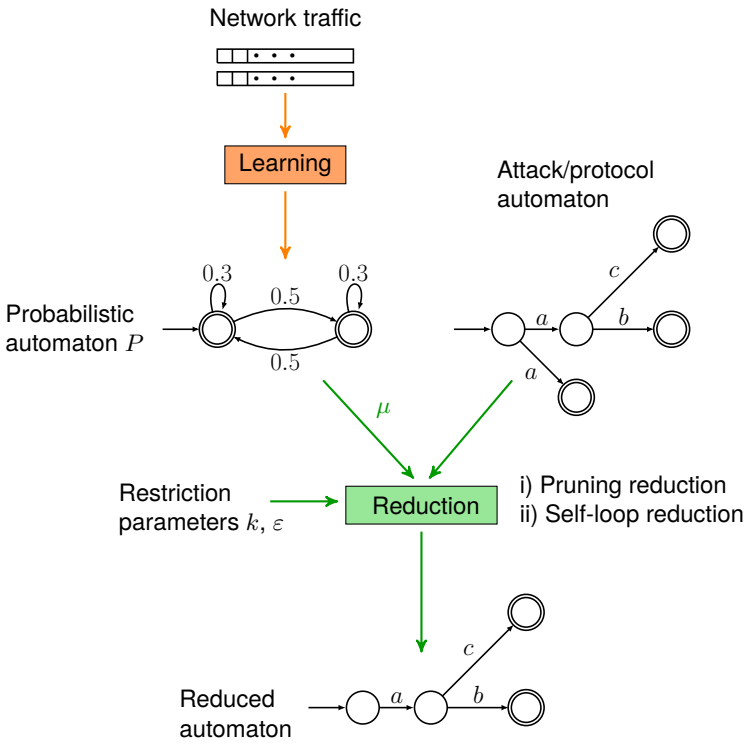


Figure 5.1: The workflow of reducing automata for network traffic filtering. In the first step, a PA P is synthesized from the input traffic. In the second step, P and an NFA describing attacks or protocols to be sought in the network traffic are the input of the reduction. The reduction yields a reduced automaton satisfying the restriction conditions.

Namely, the pruning reduction finds a set R of states and uses the restriction of the NFA to the set $Q \setminus R$ to reduce the automaton. The following lemma estimates the error (the distance between the original NFA and the reduced NFA with respect to the input PA P) when we remove all states in a set R from the input NFA. If we remove all states from R, then in the worst case, all final states reached by the states from R become non-accessible in the reduced automaton. Therefore, the total error is at most the sum of probabilities of the languages accepted in the final states reached from R. For simplicity, for an NFA $A = (Q, \Sigma, \delta, q_0, F)$, a PA P, and for each $q \in F$, we define the function $\theta_{P,A}$ as

$$\theta_{P,A}(q) = \sum_{x \in L_A^{-1}(q)} f_P(x). \tag{5.1}$$

The value $\theta_{P,A}(q)$ gives a probability that a randomly chosen string according to the distribution f_P belongs to the language $L_A^{-1}(q)$ (i.e., a probability of the backward language of q). Note that, in the following lemma, $\alpha(q)$ denotes the set of final states reached from q. More precisely, for an NFA $A = (Q, \Sigma, \delta, q_0, F)$, the function $\alpha : Q \to 2^F$ is defined as $\alpha(q) = \{q_f \in F \mid \exists w \in \Sigma^* : q_f \in \hat{\delta}(q, w)\}$ and its pointwise extension $\alpha(S) = \bigcup_{q \in S} \alpha(q)$ to a set of states S.

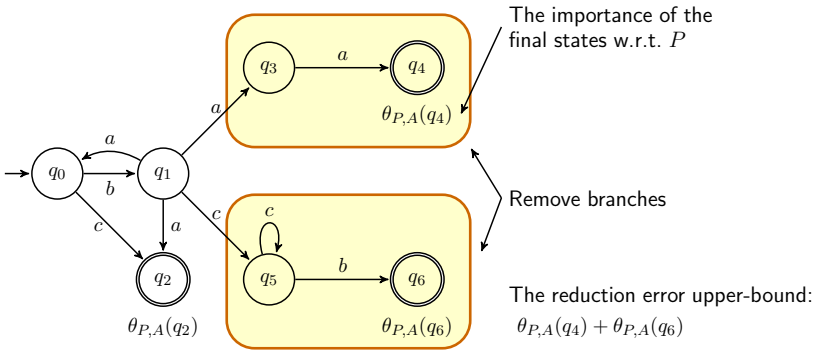


Figure 5.2: An illustration of the pruning reduction.

Lemma 26. *Let P be a PA over Σ, $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA, and $R \subseteq Q$. Further, consider the restricted automaton $A' = A_{|Q \setminus R}$. Then,*

$$d_P(L(A), L(A')) \leq \sum_{q \in \alpha(R)} \theta_{P,A}(q). \tag{5.2}$$

Proof. We start with an estimation of the symmetric difference of languages $L(A)$ and $L(A')$. Because we use automata pruning only, the language of the modified automaton A' forms a subset of the language $L(A)$. In the worst case, we remove all the final states in $\alpha(R)$, and therefore,

$$L(A) \triangle L(A') = L(A) \setminus L(A') \subseteq \bigcup_{q \in \alpha(R)} L_A^{-1}(q). \tag{5.3}$$

From the definition of the function f_P, for any $L_1, L_2 \subseteq \Sigma^*$, we get that

$$f_P(L_1 \cup L_2) \leq f_P(L_1) + f_P(L_2). \tag{5.4}$$

Further, from the definition of the distance d_P, together with Equations 5.3 and 5.4, we finally get the following:

$$\begin{aligned} d_P(L(A), L(A')) &= f_P(L(A) \triangle L(A')) \\ &\leq f_P\left(\bigcup_{q \in \alpha(R)} L_A^{-1}(q) \right) \\ &\leq \sum_{q \in \alpha(R)} \theta_{P,A}(q). \end{aligned} \tag{5.5}$$

Note that, the first inequality follows from Equation 5.3. □

From the previous lemma, we have that the error depends on the removed final states. Roughly, the pruning reduction works as follows. In the first step, we label each state q of the input automaton by $\alpha(q)$, which can be done by a simple reachability analysis. Then, the reduction finds a subset F' of the final states of an input NFA and removes all states labeled by $\alpha(q) \subseteq F'$ from the input NFA. The set F' is chosen according to the type of the performed reduction (the k-pruning reduction or the ε-pruning reduction). Now, we focus on the selection of F' and provide a more detail description of the reduction algorithms for each reduction type.

5.1.1 k-pruning Reduction

In this subsection, we deal with the k-pruning reduction. Here, the parameter k, which restricts the reduction, is the ratio of the number of states of the reduced automaton and the number of states of the input automaton. If $A = (Q, \Sigma, \delta, q_0, F)$ stands for an input automaton, then for the reduced automaton $A' = (Q', \Sigma, \delta', q_0, F')$, we have that

$$k \leq \frac{|Q'|}{|Q|}. \tag{5.6}$$

Moreover, we want the reduction to return the automaton with the set of states Q' such that the values $|Q'|/|Q|$ and k are as close as possible. In order to express the corresponding constraints on the set $F' \subseteq F$, we assume the following functions v, c and relation $\succeq$.

- The function $v : 2^F \to \mathbb{N}_0$ is defined as $v(R) = |\{q \in Q \mid \alpha(q) \subseteq R\}|$. For any $R \subseteq F$, the value $v(R)$ thus represents the number of states of the input NFA A labeled with $\alpha(q) \subseteq R$. If we decide to remove the final states in R, the value of $v(R)$ determines a total number of removed states of the input NFA. Note that, the function v considers also a possibility of removing of an initial state. In this case, the automata restriction operation adds a new initial state (our used automata formalism does not support automata with no initial state). An automaton with a single state and no final states is semantically equivalent to an empty automaton (automaton with no states).

- For a PA P over Σ, the function $c : 2^F \to \mathbb{R}$ is defined as $c(R) = \sum_{q \in \alpha(R)} \theta_{P,A}(q)$. For any $R \subseteq F$, the value $c(R)$ thus represents the error we obtain, if we remove the final states in R from the input automaton.

- Two sets $R, S \subseteq F$ are in the relation $R \succeq S$ iff $v(R) > v(S)$, or $v(R) = v(S)$ and $c(R) \leq c(S)$. For two sets $R, S \subseteq F$, we thus have $R \succeq S$ if the reduction according to R yields a smaller automaton or an automaton with the same size and the smaller error.

The k-pruning reduction thus looks for a set $R \subseteq F$ for which $v(R)$ is maximal such that $v(R) \leq W$ where $W = (1 - k)|Q|$. If there are two sets $S_1 \neq S_2$ such that $v(S_1) = v(S_2)$, we choose the set S_i with a smaller value $c(S_i)$ (i.e., we make the smaller error).

Formally, we can formulate the problem as follows. We are looking for a set $R \subseteq F$ such that $v(R) \leq W$, and for every $R' \subseteq F$ such that $v(R') \leq W$, we have $R \succeq R'$. Since we assume that the number of final states of the input automaton is substantially less than the number of all states, we use an exhaustive search in the algorithm. After we find the set R, we remove all those states r of the input automaton A that are labeled with $\alpha(r) \subseteq R$. The k-pruning reduction in pseudocode is shown in Algorithm 6.

On the 9th line of this algorithm, for the states of the reduced automaton, it holds that $|Q_r| = |Q| - v(R)$. Moreover, from the condition on the 5th line, we have that $v(R) \leq W = |Q| - k|Q|$. Together, we get $k|Q| \leq |Q_r| \leq |Q'|$. Moreover, the error that the reduction will cause can be expressed according to Lemma 26 as

$$d_P(L(A), L(A')) \leq \sum_{q \in R} \theta_{P,A}(q) \tag{5.7}$$

where R is the set obtained after the 8th line in the algorithm.

Algorithm 6: k-PRUNING REDUCTION

Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$, PA P over Σ, and $k \in [0, 1]$
Output: NFA $A' = (Q', \Sigma, \delta', q_0, F')$ such that $k \leq \frac{|Q'|}{|Q|}$

1: Compute the values of $\theta_{P,A}(q_f)$ for each $q_f \in F$
2: $R \leftarrow \emptyset$
3: $W \leftarrow (1 - k)|Q|$
4: **foreach** $S \in 2^F$ **do**
5: **if** $v(S) \leq W$ **and** $S \succeq R$ **then**
6: $R \leftarrow S$
7: **end**
8: **end**
9: $Q_r \leftarrow Q \setminus \{q \in Q \mid \alpha(q) \subseteq R\}$
10: **return** $A' = A_{|Q_r} = (Q', \Sigma, \delta', q_0, F')$

In the presented k-pruning algorithm, we use the function c (within testing $\succeq$), and for computing its value for some set, we need the value $\theta_{P,A}(q_f)$ for every final state $q_f \in F$. This value can be computed using methods described in the previous chapter. Moreover, in Section 5.4, we re-visit this issue, and show some improvements for computing $\theta_{P,A}(q_f)$.

Modifications. In some applications, it can be appropriate to specify the maximum number of states of the reduced automaton (i.e., $k \geq |Q'|/|Q|$). For instance, if we know that we are able to synthesize automata with at most n states into HW. Then, we are looking for a reduced automaton with at most n states and with the least possible error.

To formalize this modified k-pruning reduction (to distinguish, we denote this modified k-pruning reduction as the $\overline{k}$-pruning reduction), we introduce the following relation. Two sets $R, S \subseteq F$ are in the relation $R \succeq_c S$ iff $c(R) < c(S)$, or $c(R) = c(S)$ and $v(R) \geq v(S)$. If we set $W = (1 - \overline{k})|Q|$, then we are looking for a set $R \subseteq F$ such that $v(R) \geq W$, and for every $R' \subseteq F$ such that $v(R') \geq W$, we have $R \succeq_c R'$.

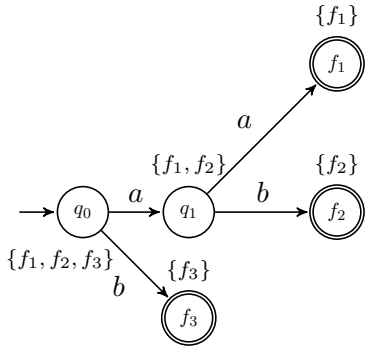
For computing the $\overline{k}$-pruning reduction, we can easily modify Algorithm 6. First, we need to modify the initial solution on the second line as $R \leftarrow F$. The second modification is the condition on 5th line, which is replaced by the $v(S) \geq W \wedge S \succeq_c R$. Note that, since our used automata formalism does not support automata with no states, if the selected set of states Q_r in the algorithm is equal to $\emptyset$, the inequality $\overline{k} \geq |Q'|/|Q|$ need not be satisfied.

Below, we illustrate the k-pruning reduction on an example.

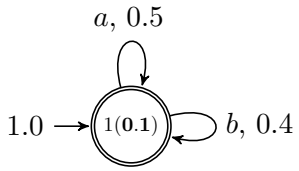
Example 27. *Consider a simple NFA and a PA given in Figure 5.3. For each state of the NFA there is a corresponding value of α. We show the k-pruning reduction according to the described algorithm on these automata. In the first step, we compute the values of $\theta_{P,A}(q_f)$ for each $q_f \in F$:*

$$\theta_{P,A}(f_1) = 0.5 \cdot 0.5 \cdot 0.1 = 0.025, \tag{5.8}$$
$$\theta_{P,A}(f_2) = 0.5 \cdot 0.4 \cdot 0.1 = 0.02, \tag{5.9}$$
$$\theta_{P,A}(f_3) = 0.4 \cdot 0.1 = 0.04. \tag{5.10}$$

Then, we are able to compute the values of functions c and v for each $R \subseteq F$, which are given in Table 5.1. If we chose the restriction parameter of the k-pruning reduction to be



(a) Input NFA A over $\{a, b\}$.



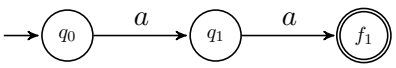
(b) Input PA P over $\{a, b\}$.

Figure 5.3: Example of an NFA and a PA for the k-pruning reduction.

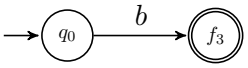
$k = \frac{1}{2}$, *then* $W = 2.5$. *Since we have* $c(\{f_1, f_3\}) > c(\{f_2, f_3\})$, *Algorithm 6 after the 8th line selects the set* $R = \{f_2, f_3\}$. *The reduced automaton* A'_1 *is then given in Figure 5.4a. If we chose the restriction parameter* $k = \frac{1}{3}$, *then* $W = 3\frac{1}{3}$, *and the set* $R = \{f_1, f_2\}$ *is selected. The reduced automaton* A'_2 *for this case is given in Figure 5.4b.*

Table 5.1: Values of the functions v and c.

subset of F	v	c
$\emptyset$	0	0.0
$\{f_1\}$	1	0.025
$\{f_2\}$	1	0.02
$\{f_3\}$	1	0.04
$\{f_1, f_2\}$	3	0.045
$\{f_1, f_3\}$	2	0.065
$\{f_2, f_3\}$	2	0.06
$\{f_1, f_2, f_3\}$	5	0.085



(a) Reduced automaton A'_1 for parameter $k = \frac{1}{2}$.



(b) Reduced automaton A'_2 for parameter $k = \frac{1}{3}$.

Figure 5.4: The results of the k-pruning reduction.

5.1.2 ε-pruning Reduction

In this subsection, we briefly describe the ε-pruning reduction. In this case, the parameter ε that restricts the reduction is the maximal probabilistic distance of the input and the reduced automaton (i.e., for an input NFA A and the reduced NFA A', it holds that $d_P(L(A), L(A')) \leq \varepsilon$).

Again, the problem can be formulated as follows. We are looking for a set $R \subseteq F$ such that $c(R) \leq \varepsilon$, and for every $R' \subseteq F$ such that $c(R') \leq \varepsilon$, we have $R \succeq R'$. For the

ε-pruning reduction, we can use a very similar algorithm as in the case of the k-pruning reduction. Our algorithm in pseudocode is shown in Algorithm 7.

Algorithm 7: ε-PRUNING REDUCTION

Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$, PA P over Σ, and $\varepsilon \geq 0$
Output: NFA $A' = (Q', \Sigma, \delta', q_0, F')$ such that $d_P(L(A), L(A')) \leq \varepsilon$

```
1: Compute the values of θ_{P,A}(q_f) for each q_f ∈ F
2: R ← ∅
3: W ← ε
4: foreach S ∈ 2^F do
5:     if c(S) ≤ W and S ⪰ R then
6:         R ← S
7:     end
8: end
9: Q_r ← Q \ {q ∈ Q | α(q) ⊆ R}
10: return A' = A_{|Q_r} = (Q', Σ, δ', q_0, F')
```

The functions v, c, and the relation $\succeq$ are defined as in the case of the k-pruning reduction. The fact that $d_P(L(A), L(A')) \leq \varepsilon$ follows from Lemma 26.

5.2 Optimization

In this section, we look at some optimizations and heuristics that can be used for the pruning reduction. We introduce two possible ways of optimization. The first one is based on a reduction of the search space in Algorithms 6 and 7. The second one uses a structure of the automata used in the network traffic filtering.

5.2.1 Search Space Reduction

The first reduction we consider aims at reduction of the search space 2^F in Algorithms 6 and 7. The reduction of the search space is based on the observation that if we find some $M \in 2^F$ such that $v(M) > W$ or $c(M) > W$, respectively, then we do not need to check $v(M') > W$ or $c(M') > W$ for any $M' \supset M$. This optimization can be used in the main loop of each algorithm to save a few evaluations of the function v. Another possible optimization is to compute $v(\{q_f\})$ or $c(\{q_f\})$, respectively, for each $q_f \in F$ before the main loop, find the set $F' = \{q_f \in F \mid v(\{q_f\}) \leq W\}$ or $F' = \{q_f \in F \mid c(\{q_f\}) \leq W\}$, respectively, and iterate only through the set $2^{F'}$ (not 2^F).

5.2.2 Subautomata Pruning

So far, we have been looking for some $R \subseteq F$ satisfying our constraints given by the type of the pruning reduction. However, if the number of final states is, for instance, 100, the total search space is about 10^{30}, which is hardly practically solvable. Therefore, we give up the requirement of optimality. For that, we use the structure of the automata. We identify independent subautomata in the structure of the input automaton and we will remove the entire selected subautomata. A common structure of the automata used in network traffic filtering is shown in Figure 5.5. An automaton with such a structure can be divided into several independent subautomata $A_1, \ldots, A_n$, with the corresponding sets of final states

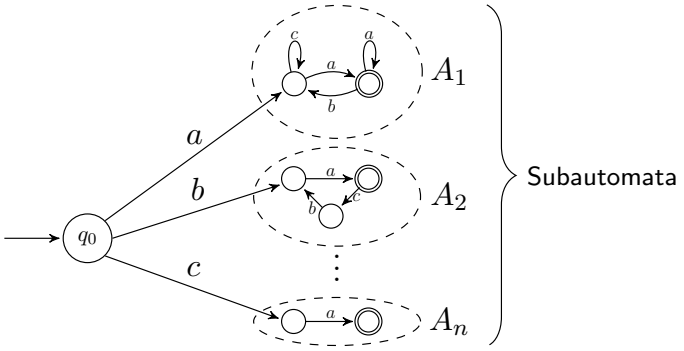


Figure 5.5: A common structure of NFAs used in the network traffic filtering. The whole automaton can be divided into n independent subautomata.

$F_1, \ldots, F_n$. Clearly, these sets are subsets of F. We can thus perform a *relaxed pruning reduction*, which consists in removing the whole subautomata.

The subautomata are independent, therefore, if we are not removing all the final states (i.e., $F \neq F_i \cup F_j$), it holds that $v(F_i) + v(F_j) = v(F_i \cup F_j)$, and $c(F_i) + c(F_j) = c(F_i \cup F_j)$ for $i \neq j$. Further, we can define vectors $\boldsymbol{v} \in \mathbb{R}^n$ and $\boldsymbol{c} \in \mathbb{R}^n$ whose elements are given by $\boldsymbol{v}[i] = v(F_i)$, $\boldsymbol{c}[i] = c(F_i)$, respectively, for $i \in \{1, \ldots, n\}$. The *relaxed ε-pruning reduction* can thus be formulated as follows:

$$\begin{aligned} \text{maximize} \quad & \boldsymbol{v}^\top \cdot \boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{c}^\top \cdot \boldsymbol{x} \leq \varepsilon \\ & \boldsymbol{x} \in \{0,1\}^n \end{aligned} \tag{5.11}$$

where $\boldsymbol{x} = (x_1, \ldots, x_n)^\top$ contains the variables to be optimized. If $x_i = 1$, then the subautomaton A_i is removed from the input NFA. Vector $\boldsymbol{x}$ therefore encodes the information about which subautomata are removed. The total error, when we remove the selected subautomata (encoded in vector $\boldsymbol{x}$), must be less then ε, i.e., a condition $\boldsymbol{c}^\top \cdot \boldsymbol{x} \leq \varepsilon$ must be satisfied. On the other hand we want to remove as many subautomata as possible. For this reason, we maximize the value of $\boldsymbol{v}^\top \cdot \boldsymbol{x}$.

A problem of the form

$$\begin{aligned} \text{maximize} \quad & \boldsymbol{u}^\top \cdot \boldsymbol{x} \\ \text{subject to} \quad & \mathbf{A} \cdot \boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{Z}^n \end{aligned} \tag{5.12}$$

where $\boldsymbol{u} = (u_1, \ldots, u_n)^\top$ is a vector, $\mathbf{A}$ is an $m \times n$ matrix, and $\boldsymbol{b} = (b_1, \ldots, b_m)^\top$ is a vector, and the vector $\boldsymbol{x} = (x_1, \ldots, x_n)^\top$ contains the variables to be optimized, is called a *(pure) integer linear program* [9]. If each variable can only take the value of 0 or 1, the problem is called a *0,1 integer linear program*. Integer linear programming can be viewed as a special case of general linear programming (a linear program is formulated as the integer linear program with the only difference that the variables can take arbitrary nonnegative real values). Although, a linear programming problem is solvable in polynomial time [34], it was shown that the integer programming is NP-hard [9]. However, there are still algorithms and heuristics that allow us to solve integer linear programming problems in practice (e.g., the branch-and-bound method, the cutting plane method, and others [9]).

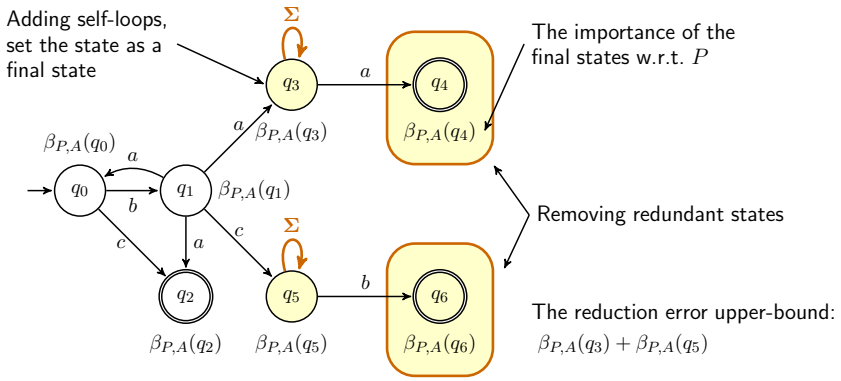


Figure 5.6: An illustration of the self-loop reduction.

In a similar way, we can formulate a *relaxed k-pruning reduction* as:

$$\begin{aligned} \text{maximize} \quad & \boldsymbol{v}^\top \cdot \boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{v}^\top \cdot \boldsymbol{x} \leq W \\ & \boldsymbol{x} \in \{0,1\}^n. \end{aligned} \tag{5.13}$$

Moreover, if such a modified problem has more feasible solutions $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$, we select the one with the least error, i.e., $\boldsymbol{x} \in \arg\min_{\boldsymbol{x} \in X}\{\boldsymbol{c}^\top \cdot \boldsymbol{x}\}$. The meaning of the vector $\boldsymbol{x}$ is the same as in the case of the relaxed ε-pruning reduction.

In Section 5.1.1, we introduced the modified $\overline{k}$-pruning reduction. For completeness, we introduce here a relaxed $\overline{k}$-pruning reduction as well. The *relaxed $\overline{k}$-pruning reduction* can be formulated in the sense of integer programming as

$$\begin{aligned} \text{minimize} \quad & \boldsymbol{c}^\top \cdot \boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{v}^\top \cdot \boldsymbol{x} \geq W \end{aligned} \tag{5.14}$$

where $\boldsymbol{x} \in \{0,1\}^n$ and the other vectors have the same meaning as in the previous cases.

5.3 Self-loop Reduction

In this section, we present our second approach for reducing finite automata. As in the case of the pruning reduction, the self-loop reduction is a language non-preserving reduction. The self-loop reduction consists of adding self-loops to certain states (and making these states final), followed by removing all other transitions from these states and trimming the modified automaton. The choice of states for adding self-loops depends again on the input probabilistic automaton. The reduction is restricted by the parameter (k or ε) with the same meaning as in the case of the pruning reduction. Therefore, we again divide the self-loop reduction into the k-self-loop reduction and the ε-self-loop reduction. An illustration of the self-loop reduction is shown in Figure 5.6.

Further, for a PA $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a \in \Sigma})$ and a string $w \in \Sigma^*$, we define the *reachability weight of w in P* as

$$\mathsf{weight}_P(w) = \boldsymbol{\alpha}_0^\top \cdot \mathbf{A}_w \cdot \mathbf{1} \tag{5.15}$$

where $\mathbf{1}$ is the vector of 1's. Its pointwise extension to a language L is defined as $\mathsf{weight}_P(L) = \sum_{w \in L} \mathsf{weight}_P(w)$. Further, for a PA P, an NFA $A = (Q, \Sigma, \delta, q_0, F)$, and a state $q \in Q$,

we define the function $\beta_{P,A}$ as

$$\beta_{P,A}(q) = \sum_{w \in L_A^{-1}(q)} \mathsf{weight}_P(w). \tag{5.16}$$

Note that, we use $w.L$ as an abbreviation for $\{w\}.L$.

First, we formally describe the operation of adding self-loops. Assume an operation $\mathsf{sladd}(A, Q')$ modifying an input NFA A by adding self-loops for every state in Q'. Self-loops are added for every symbol from an alphabet Σ. Formally, for an NFA $A = (Q, \Sigma, \delta, q_0, F)$ and a set of states $Q' \subseteq Q$, the operation sladd is defined as $\mathsf{sladd}(A, Q') = \mathsf{trim}(A')$ where $A' = (Q, \Sigma, \delta', q_0, F \cup Q')$ is an NFA whose transition function δ' is defined, for all $p \in Q$ and $a \in \Sigma$, as

$$\delta'(p, a) = \begin{cases} \{p\} & \text{if } p \in Q', \\ \delta(p, a) & \text{otherwise.} \end{cases} \tag{5.17}$$

Now, we give a lemma, which is used in the following theorem. The proof of this lemma is given in Appendix A.

Lemma 28. *Let P be a trim PA over Σ, and $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Then, for all $q \in Q$ we have*

$$f_P(L_A^{-1}(q).\Sigma^*) \leq \beta_{P,A}(q). \tag{5.18}$$

The following theorem establishes the maximal error (i.e., the distance between the input NFA and the reduced NFA) that we get if we add self-loops for the states in the set Q'. This theorem uses helpful lemmas, which are given in Appendix A.

Theorem 29. *Let P be a PA over Σ, $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA, and $Q' \subseteq Q$ be a set of states of A. Then, the following inequality holds:*

$$d_P(L(A), L(A')) \leq \sum_{q \in Q'} \beta_{P,A}(q) \tag{5.19}$$

where $A' = \mathsf{sladd}(A, Q')$.

Proof. First, we remark that a self-loop addition causes a language over-approximation: After we reach some state from Q', we accept everything. Therefore, the symmetric difference of languages $L(A)$ and $L(A')$ can be bounded as

$$L(A) \triangle L(A') = L(A') \setminus L(A) \subseteq \bigcup_{q \in Q'} L_A^{-1}(q).\Sigma^*. \tag{5.20}$$

Hence, the following inequalities hold:

$$d_P(L(A), L(A')) \leq f_P\left(\bigcup_{q \in Q'} L_A^{-1}(q).\Sigma^*\right) \leq \sum_{q \in Q'} f_P(L_A^{-1}(q).\Sigma^*). \tag{5.21}$$

Finally, using Lemma 28, we get Equality 5.19. □

Now, we can introduce algorithms for each type of the self-loop reduction. However, before that, we state an important observation. Consider a trim NFA A and a set $Q' \subseteq Q_A$. Further, let NFA $A' = \mathsf{sladd}(A, Q')$. If $q \notin Q_{A'}$, then $\mathsf{sladd}(A, Q') = \mathsf{sladd}(A, Q' \cup \{q\})$. This holds due to **1.** Adding self-loops and changing the acceptance mode of an non-accessible state does not alter the language **2.** Removing outgoing transitions from q cannot make any new state non-accessible since if the only way to reach such a state had been through q, then such a state would have been non-accessible even before due to non-accessibility of q itself. This observation will be used for a more precise estimation of the reduction error.

5.3.1 ε-self-loop Reduction

The meaning of the restriction parameter ε is the same as in the case of the ε-pruning reduction (i.e., it is the maximal probabilistic distance of the input and the reduced automaton). The optimal selection of the states where self-loops will be added may be computationally difficult. Therefore, we introduce a more efficient greedy algorithm. Our greedy algorithm selects states where self-loops will be added according to increasing values of $\beta_{P,A}$. In the algorithm, the values $\beta_{P,A}(q)$ are first computed for each state q of the input NFA. The exact methods for computing these values are described in Section 5.4. In the algorithm, we use a set V for storing states to which self-loops are added in the end. We also use a set R for storing all processed states.

We iterate until we process all states from Q. In every iteration, we choose a state q from $Q \setminus R$ with the least value of $\beta_{P,A}$, and if the error of adding self-loops to states from the set $V \cup \{q\}$ is less than or equal to ε, we add this state to V.

Although, the maximal error for a set $V \subseteq Q$ is bounded by the inequality in Theorem 29, we can obtain a more accurate estimation. The error estimation is performed in the procedure computeError. In this procedure, we add self-loops to automaton A and obtain a reduced automaton A' (i.e., we perform $\mathsf{sladd}(A, V)$). The error then does not depend on all states from V but only on states from $Q_{A'} \cap V$ (some states are removed due to a self-loop is added to their predecessor). The error is then given as the sum of the $\beta_{P,A}$ values of all states from this intersection. The algorithm in pseudocode is shown in Algorithm 8.

Algorithm 8: ε-SELF-LOOP REDUCTION

Input: Trim NFA $A = (Q, \Sigma, \delta, q_0, F)$, PA P over Σ, and $\varepsilon \geq 0$
Output: NFA $A' = (Q', \Sigma, \delta', q_0, F')$ such that $d_P(L(A), L(A')) \leq \varepsilon$

```
1:  Compute the values of β_{P,A}(q) for each q ∈ Q
2:  V ← ∅
3:  R ← ∅
4:  while R ≠ Q do
5:      q' ← arg min_{q∈Q\R}{β_{P,A}(q)}
6:      R ← R ∪ {q'}
7:      e ← computeError(A, V ∪ {q'})
8:      if e ≤ ε then
9:          V ← V ∪ {q'}
10:     end
11: end
12: return A' = sladd(A, V)
```

Procedure: computeError(A, V)

```
1:  A' ← sladd(A, V)
2:  W ← Q_{A'} ∩ V
3:  err ← Σ_{q∈W} β_{P,A}(q)
4:  return err
```

From the observation below Theorem 29, we get that $\mathsf{sladd}(A, V) = \mathsf{sladd}(A, W) = A'$ (in the procedure computeError), and therefore, according to Theorem 29, we obtain inequality

$$d_P(L(A), L(A')) \leq \sum_{q \in W} \beta_{P,A}(q) = err. \tag{5.22}$$

Hence, $d_P(L(A), L(A')) \leq \varepsilon$. Although, our algorithm returns a reduced automaton with the error less than or equal to a given ε, the choice of the final set V might not be optimal.

5.3.2 k-self-loop Reduction

Now, we focus on the k-self-loop reduction. Recall that, the restriction parameter k sets the maximal ratio of the number of states of the reduced automaton and the number of states of the input automaton. Moreover, we want the reduction to return the automaton with the set of states Q' such that the values $|Q'|/|Q|$ and k are as close as possible. Our k-self-loop reduction algorithm is greedy as in the case of the ε-self-loop reduction. We again use a set R for storing all processed states and a set V to store states that are then used for adding self-loops. States from the set $Q \setminus R$ are picked out in ascending order of their $\beta_{P,A}$ values. Picking states in ascending order of their $\beta_{P,A}$ values is a heuristics, which tries to minimize the reduction error. In every iteration, the reduced automaton A' is created, and if the number of its states is less than or equal to lim, we add the picked state into V. The algorithm in pseudocode is shown in Algorithm 9.

Algorithm 9: k-SELF-LOOP REDUCTION

Input: Trim NFA $A = (Q, \Sigma, \delta, q_0, F)$, PA P over Σ, and $k \in [0, 1]$
Output: NFA $A' = (Q', \Sigma, \delta', q_0, F')$ such that $k \leq \frac{|Q'|}{|Q|}$

Compute the values of $\beta_{P,A}(q)$ for each $q \in Q$
$V \leftarrow \emptyset$, $R \leftarrow \emptyset$
$e \leftarrow 0$
$lim \leftarrow k \cdot |Q|$
while $R \neq Q$ **do**
    $q' \leftarrow \arg\min_{q \in Q \setminus R}\{\beta_{P,A}(q)\}$
    $R \leftarrow R \cup \{q'\}$
    $A' \leftarrow \mathsf{sladd}(A, V \cup \{q'\})$
    $e \leftarrow |Q_{A'}|$
    if $e \geq lim$ **then**
        $V \leftarrow V \cup \{q'\}$
    end
end
return $A' = \mathsf{sladd}(A, V)$

Our algorithm returns a reduced NFA with at least $k \cdot |Q|$ states. However, the set V need not be chosen optimally to the error. There can be some other set V' such that the NFA reduced according to the set V' has the same number of states as the NFA reduced according to the set V. Moreover, the error caused by the reduction according to V' can be less than the error caused by the reduction according to V.

Modifications. As in the case of the $\overline{k}$-pruning reduction, we can formulate the $\overline{k}$-self-loop reduction, so that the number of states of the reduced automaton satisfy $\overline{k} \geq |Q'|/|Q|$; and, moreover, we try to find a reduced automaton with the least possible error.

For the $\overline{k}$-self-loop reduction, we can use modified greedy Algorithm 9. We want to find a reduced automaton with the least error, and so we need two new variables—*err* and *bestErr*. The variable *bestErr* is initialized to ∞. In each iteration of the main loop, the error obtained in that iteration is computed, i.e., $err \leftarrow \mathsf{computeError}(A, V \cup \{q'\})$. Then, a new set V is accepted if $e \leq lim \ \wedge \ err \leq bestErr$. Altogether, we replace lines 10–12 in the main loop by the following:

```
1: err ← computeError(A, V ∪ {q'})
2: if e ≤ lim and err ≤ bestErr then
3:     bestErr ← err
4:     V ← V ∪ {q'}
5: end
```

Since we initialize the value of *bestErr* to ∞, when the condition $e \leq lim$ is satisfied for the first time, we accept the corresponding set $V \cup \{q'\}$ where q' is the chosen state. This way, we set our initial solution, which can be improved in the next iterations.

Example 30. *Consider the same automata as in the case of the example of the pruning reduction (given in Figure 5.3). We show the reduction on these automata using the described algorithm for the ε-self-loop reduction (Alg. 8). In the first step, we compute the $\beta_{P,A}$ value for each state:*

$$\beta_{P,A}(q_0) = 1 \cdot 1 = 1, \tag{5.23}$$
$$\beta_{P,A}(q_1) = 1 \cdot 0.5 \cdot 1 = 0.5, \tag{5.24}$$
$$\beta_{P,A}(f_1) = 1 \cdot 0.5 \cdot 0.5 \cdot 1 = 0.25, \tag{5.25}$$
$$\beta_{P,A}(f_2) = 1 \cdot 0.5 \cdot 0.4 \cdot 1 = 0.2, \tag{5.26}$$
$$\beta_{P,A}(f_3) = 1 \cdot 0.4 \cdot 1 = 0.4. \tag{5.27}$$

Then, we use Algorithm 8 for the reduction. We set the restriction parameter to $\varepsilon = 0.5$. Each step of the algorithm and the corresponding values of variables are shown in Table 5.2.

Table 5.2: Values of variables for each step of the ε-self-loop reduction algorithm. Note that, by the variable *err*, we denote the value $\mathsf{computeError}(A, V)$.

step	q'	R	e	V	err
0	–	$\emptyset$	–	$\emptyset$	0
1	f_2	$\{f_2\}$	0.2	$\{f_2\}$	0.2
2	f_1	$\{f_1, f_2\}$	0.45	$\{f_1, f_2\}$	0.45
3	f_3	$\{f_1, f_2, f_3\}$	0.85	$\{f_1, f_2\}$	0.45
4	q_1	$\{f_1, f_2, f_3, q_1\}$	0.5	$\{f_1, f_2, q_1\}$	0.5
5	q_0	Q	1.0	$\{f_1, f_2, q_1\}$	0.5

The result of the reduction is then shown in Figure 5.7. The exact reduction error (the probabilistic distance) is given as $d_P(L(A), L(A')) = 0.455$.

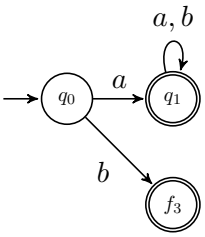


Figure 5.7: Automaton A' reduced from the automaton in Figure 5.3 using Algorithm 8 with the parameter $\varepsilon = 0.5$.

5.4 State Labels Computation

In the previous sections, we described two ways of reducing automata. The pruning reduction labels each final state q_f of the input NFA A by the value $\theta_{P,A}(q_f)$ where P is the input PA. The self-loop reduction then labels each state q of the input NFA by the value $\beta_{P,A}(q)$. In this section, we look in more details on the ways how efficiently compute these values.

5.4.1 Computing the Values of $\theta_{P,A}$

We start with the computation of the value of $\theta_{P,A}(q)$, which is used for the pruning reduction. We give two approaches: The first can be used for general NFAs, and the other, more efficient, can be used for UFAs only. We start with an algorithm for computing $\theta_{P,A}(q)$ for general NFAs. The algorithm is given in Algorithm 10.

In the algorithm, we use the operation $\mathsf{backSubautomaton}(A, q)$ that, for an NFA A and a state q, returns an NFA accepting the language $L_A^{-1}(q)$. This operation sets q as the only final state in A and performs trimming of this modified automaton. Other operations that are used, are $\mathsf{isUFA}(A)$ and $\mathsf{disambiguation}(A)$ for an NFA A. The former one checks whether an NFA A is unambiguous. This operation is a straightforward implementation of Theorem 25. The latter operation performs disambiguation on the NFA A (based on Algorithm 5).

Algorithm 10: Computation of values of $\theta_{P,A}$

Input: PA $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ over Σ, NFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: Vector $\boldsymbol{\gamma}$ where $\boldsymbol{\gamma}[q_f] = \theta_{P,A}(q_f)$ for each $q_f \in F$

1: **foreach** $q \in F$ **do**
2: $\quad A_q \leftarrow \mathsf{backSubautomaton}(A, q)$
3: $\quad$ **if not** $\mathsf{isUFA}(A_q)$ **then**
4: $\quad\quad A_q \leftarrow \mathsf{disambiguation}(A_q)$
5: $\quad$ **end**
6: $\quad P' \leftarrow \mathsf{trim}(P \odot A_q) = (\boldsymbol{\nu}_0, \boldsymbol{\nu}_f, \{\mathbf{N}_a\}_{a\in\Sigma})$
7: $\quad \boldsymbol{\gamma}[q] \leftarrow \boldsymbol{\nu}_0^\top (\mathbf{I} - \mathbf{N}_\Sigma)^{-1} \boldsymbol{\nu}_f$
8: **end**
9: **return** $\boldsymbol{\gamma}$

The below theorem proves correctness of the algorithm.

Theorem 31. *Algorithm 10 is correct.*

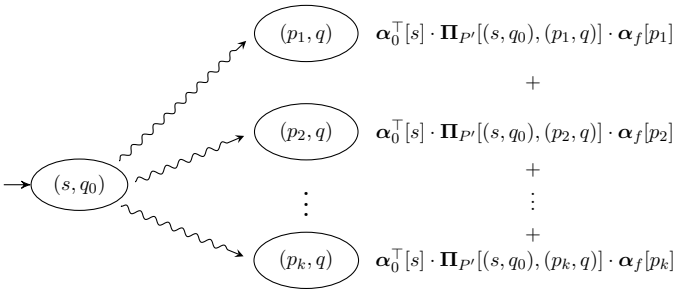


Figure 5.8: The basic intuition behind Theorem 32.

Proof. Let us consider some $q \in F$. Then, the backward language of the NFA A_q on line 2 is given as $L(A_q) = L_A^{-1}(q)$. Further, after the 5th line, A_q is an unambiguous automaton. Therefore, from Lemma 21, we get $f_P(L(A_q)) = f_{P'}(\Sigma^*)$. Moreover, according to Theorems 22 and 14, we obtain

$$f_P(L(A_q)) = f_{P'}(\Sigma^*) = \boldsymbol{\nu}_0^\top (\mathbf{I} - \mathbf{N}_\Sigma)^{-1} \boldsymbol{\nu}_f. \tag{5.28}$$

Finally, because $L(A_q) = L_A^{-1}(q)$, we get

$$f_P(L(A_q)) = \theta_{P,A}(q) = \boldsymbol{\nu}_0^\top (\mathbf{I} - \mathbf{N}_\Sigma)^{-1} \boldsymbol{\nu}_f, \tag{5.29}$$

which concludes the proof. □

The advantage of the presented algorithm is that it can be used for computing the values of $\theta_{P,A}(q)$ for general NFAs. The disadvantage is the time complexity. This is caused by the repeated computation of the $\odot$-product and the inverse $(\mathbf{I} - \mathbf{N}_\Sigma)^{-1}$. Therefore, we give a theorem that allow us to compute $\theta_{P,A}(q)$ values more efficiently. This improvement, however, works only if the input automaton is unambiguous. Fortunately, as our experimental experience shows, many automata used in network traffic filtering are given as UFAs, and so the technique is useful in many cases.

The improved approach for the computation of the value of $\theta_{P,A}(q)$ is based on Theorem 32. The below theorem uses the matrix $\boldsymbol{\Pi}_{P'}$ computed from the automaton P' for a computation of $\theta_{P,A}(q)$ for each $q \in F$. The basic intuition behind the theorem is the following: The value of $\boldsymbol{\Pi}_{P'}[a, b]$ expresses the total weight (without the initial and the final weight) of all strings that label some path from the state a to the state b in P'. The states of P' are pairs of the form (p, q) where $p \in Q_P$, $q \in Q_A$. Therefore, if we want to compute $\theta_{P,A}(q)$, we find all states of the form (p, q) in the automaton P'. Then, for each this state (p, q) and an initial state (s, q_0), we compute the value $\boldsymbol{\alpha}_0^\top[s] \cdot \boldsymbol{\Pi}_{P'}[(s, q_0), (p, q)] \cdot \boldsymbol{\alpha}_f[p]$, which express the probability of accepting the strings that labels some path from the state (s, q_0) to the state (p, q). Finally, we sum all these values to obtain the value of $\theta_{P,A}(q)$.

The basic intuition behind the theorem is shown in Figure 5.8 as well. The figure shows an initial state (s, q_0) and states of the form (p_i, q) of the automaton P'. The value of $\theta_{P,A}(q)$ is given by the sum of values related to each state of the form (p_i, q).

Theorem 32. *Let $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a \in \Sigma})$ be a trimmed PA over Σ, let $A = (Q, \Sigma, \delta, q_0, F)$ be a trimmed UFA, and let $\{\mathbf{T}_a\}_{a \in \Sigma}$ be the transition matrices of* $\mathsf{weighted}(A)$*. Further, let P' be a WFA given as $P' = \mathsf{trim}(P \odot A) = (\boldsymbol{\beta}_0, \boldsymbol{\beta}_f, \{\mathbf{B}_a\}_{a \in \Sigma})$. Then, for each $q \in F$, we have*

$$\theta_{P,A}(q) = \boldsymbol{\beta}_0^\top \cdot \boldsymbol{\Pi}_{P'} \cdot \boldsymbol{\gamma}_q^{\boldsymbol{\alpha}_f} \tag{5.30}$$

where $\mathbf{\Pi}_{P'} = (\mathbf{I} - \mathbf{B}_\Sigma)^{-1} = \sum_{t \geq 0} \mathbf{B}_\Sigma^t$*, and, for each* $(p', q') \in Q_{P'}$*,*

$$\boldsymbol{\gamma}_q^{\boldsymbol{\alpha}_f}[(p', q')] = \begin{cases} \boldsymbol{\alpha}_f[p'] & \text{if } q' = q, \\ 0 & \text{otherwise.} \end{cases} \tag{5.31}$$

A proof of this theorem is given in Appendix A. The theorem can be straightforwardly used for computing the values of $\theta_{P,A}$ for a UFA A. In the first step, we compute the product WFA $P' = \mathsf{trim}(P \odot A) = (\boldsymbol{\beta}_0, \boldsymbol{\beta}_f, \{\mathbf{B}_a\}_{a \in \Sigma})$. In the second step, we compute the matrix $\mathbf{\Pi}_{P'} = (\mathbf{I} - \mathbf{B}_\Sigma)^{-1}$. Finally, we compute the values of $\theta_{P,A}(q)$ for all $q \in F$, according to Equation 5.30. For the unambiguous input automaton, this approach is more efficient than the previous algorithm because the $\odot$-product and the matrix inverse are computed only once.

5.4.2 Computing the Values of $\beta_{P,A}$

Now, we look at computing the values of $\beta_{P,A}$, which are used for the self-loop reduction. We give, as in Section 5.4.1, two approaches for the computation. The first one can be used for general NFAs, while the second, more efficient can be used for UFAs only. Note that, in the first algorithm, we assume that all states of the given PA are final (i.e., every state has nonzero probability of accepting).

We start with an algorithm for computing $\beta_{P,A}(q)$ for each state q of a general NFA A. The algorithm is given in Algorithm 11. In the algorithm, we use the same operations as in Subsection 5.4.1.

Algorithm 11: Computation of values of $\beta_{P,A}$

Input: PA $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a \in \Sigma})$ over Σ having all states final, NFA $A = (Q, \Sigma, \delta, q_0, F)$

Output: Vector $\boldsymbol{\beta}$ where $\boldsymbol{\beta}[q] = \beta_{P,A}(q)$ for each $q \in Q$

1: **foreach** $q \in Q$ **do**
2: $\quad A_q \leftarrow \mathsf{backSubautomaton}(A, q)$
3: $\quad$ **if not** $\mathsf{isUFA}(A_q)$ **then**
4: $\quad\quad A_q \leftarrow \mathsf{disambiguation}(A_q)$
5: $\quad$ **end**
6: $\quad P' \leftarrow \mathsf{trim}(P \odot A_q) = (\boldsymbol{\nu}_0, \boldsymbol{\nu}_f, \{\mathbf{N}_a\}_{a \in \Sigma})$
7: $\quad \boldsymbol{\gamma}_f[i] \leftarrow \begin{cases} 1 & \text{if } \boldsymbol{\nu}_f[i] > 0, \\ 0 & \text{otherwise,} \end{cases}$ for all $1 \leq i \leq len(\boldsymbol{\nu}_f)$
8: $\quad \boldsymbol{\beta}[q] \leftarrow \boldsymbol{\nu}_0^\top (\mathbf{I} - \mathbf{N}_\Sigma)^{-1} \boldsymbol{\gamma}_f$
9: **end**
10: **return** $\boldsymbol{\beta}$

The below theorem proves correctness of the algorithm.

Theorem 33. *Algorithm 11 is correct.*

Proof. Let us consider some $q \in Q$. Then, the backward language of the NFA A_q on line 2 is given as $L(A_q) = L_A^{-1}(q)$. Further, after the 5th line, A_q is an unambiguous automaton. Therefore, from Lemma 40, we get

$$\mathsf{weight}_P(L(A_q)) = \boldsymbol{\nu}_0^\top (\mathbf{I} - \mathbf{N}_\Sigma)^{-1} \boldsymbol{\gamma}_f. \tag{5.32}$$

Finally, since $L(A_q) = L_A^{-1}(q)$, we get

$$\mathsf{weight}_P(L(A_q)) = \mathsf{weight}_P(L_A^{-1}(q)) = \beta_{P,A}(q), \tag{5.33}$$

which concludes the proof. □

The advantage of the presented algorithm is that it can be used for computing the values of $\beta_{P,A}$ for general NFAs. Its disadvantage is again the time complexity. This is caused by the repeated computation of the $\odot$-product and the inverse $(\mathbf{I} - \mathbf{N}_\Sigma)^{-1}$. Therefore, we again give a theorem that allows us to compute $\beta_{P,A}$ values more efficiently. This theorem is an analogy to Theorem 32. This improvement, however, works only if the input automaton is unambiguous (fortunately, it can still be useful in practice). The basic intuition behind the below theorem is very similar to the intuition behind Theorem 32.

Theorem 34. *Let $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ be a trimmed DPA over Σ with just one initial state s, i.e., $I_P = \{s\}$, such that $L(\mathsf{supp}(P)) = \Sigma^*$. Further, let $A = (Q, \Sigma, \delta, q_0, F)$ be a trimmed UFA, and let $\{\mathbf{T}_a\}_{a\in\Sigma}$ be the transition matrices of $\mathsf{weighted}(A)$. We also consider WFA P' given as $P' = \mathsf{trim}(P \odot A) = (\boldsymbol{\beta}_0, \boldsymbol{\beta}_f, \{\mathbf{B}_a\}_{a\in\Sigma})$. Then, for all $r \in Q$, we have*

$$\beta_{P,A}(r) = \boldsymbol{\beta}_0^\top \cdot \boldsymbol{\Pi}_{P'} \cdot \boldsymbol{\gamma}_r^{\mathbf{1}} \tag{5.34}$$

where $\boldsymbol{\Pi}_{P'} = (\mathbf{I} - \mathbf{B}_\Sigma)^{-1} = \sum_{t\geq 0} \mathbf{B}_\Sigma^t$, and, for each $(q', r') \in Q_{P'}$,

$$\boldsymbol{\gamma}_r^{\mathbf{1}}[(q', r')] = \begin{cases} 1 & \text{if } r' = r, \\ 0 & \text{otherwise.} \end{cases} \tag{5.35}$$

The proof of this theorem is given in Appendix A. This theorem can be straightforwardly used for computing $\beta_{P,A}$ values for a UFA A. The method is almost the same as in the case of $\theta_{P,A}(q)$, but we repeat it here for completeness. In the first step, we compute the product WFA $P' = \mathsf{trim}(P \odot A) = (\boldsymbol{\beta}_0, \boldsymbol{\beta}_f, \{\mathbf{B}_a\}_{a\in\Sigma})$. In the second step, we compute the matrix $\boldsymbol{\Pi}_{P'} = (\mathbf{I} - \mathbf{B}_\Sigma)^{-1}$. Finally, we compute the values of $\beta_{P,A}$ for all $q \in Q$ using Equation 5.34. For the input unambiguous automaton, this approach is more efficient than the previous algorithm because the $\odot$-product and the matrix inverse are computed only once.

5.5 Complexity

In this section, we give an estimation of the complexity of the proposed reduction algorithms. For further consideration, we assume that the alphabet has some fixed size. The complexity of each algorithm depends on the complexity of computation of the state labels. If the input NFA A is unambiguous and the input probabilistic automaton P meets the conditions given in Theorem 32 and Theorem 34, respectively, then the state labels can be computed with complexity $\mathcal{O}(|Q_P|^3|Q_A|^3)$ (see Section 4.4). If the input NFA is ambiguous, the state labels are computed with exponential-time complexity in the number of states of A, which is caused by disambiguation.

When the state labels are already computed, the pruning reduction iterates over all subsets of F_A. Therefore, the worst-case complexity is exponential in the number of states of A (the same conclusion holds also for the modified pruning reduction and the relaxed pruning reduction).

Now, we focus on the complexity of the self-loop reduction. Meanwhile, we do not consider the complexity of computing the state labels. Before the main cycle, we can sort the states according to the value of $\beta_{P,A}$ with complexity $\mathcal{O}(|Q_A| \log |Q_A|)$. Inside the main cycle, the operation sladd can be computed in linear time in the size of A (linear in the number of states and the number of transitions), i.e., the complexity is $\mathcal{O}(|Q_A|^2)$. Hence, the complexity of the self-loop reduction without the computation of the state labels is $\mathcal{O}(|Q_A|^3)$.

To sum it up, the worst-case complexity of the pruning reduction is exponential in the number of states of A (regardless of whether or not A is unambiguous). In the case of the self-loop reduction, if the input NFA A is unambiguous and the input probabilistic automaton P meets the conditions given in Theorem 34, then the algorithm runs in polynomial time. If the conditions are not met, the worst-case complexity is exponential.

Chapter 6

Implementation

In this chapter, we provide a brief description of a prototype tool for approximate reduction of automata, which has been implemented based on the ideas presented in the previous chapters. The tool is designed as a console application. We give a basic overview of the implemented tool and describe some interesting parts in more detail.

The whole tool is implemented in Python 2.7. This language is chosen because of its expressiveness and possibility of rapid development. The implementation also uses several libraries, including:

- `NumPy`[1] (part of `SciPy`) for linear algebra operations. This library provides data types for representing vectors and matrices and operations with them (multiplication, inverse computation, eigenvectors, etc.).
- `PuLP`[2] for solving (integer) linear programming problems.
- `FAdo`[3] for operations with finite automata.

Since the `FAdo` library does not provide disambiguation and since we also need more control over the automata representation and operations, the classes for representing of WFAs (NFAs) and the necessary operations over them were implemented. However, for compatibility, our representation of NFAs can be converted to the FAdo format.

Below, we give more details on the implemented tool. First, we present the architecture of the proposed tool. Then, we describe the basic classes, and we provide some implementation details related to the probabilistic distance computation and the approximate reduction.

6.1 The Architecture of the Proposed Tool

In this section, we focus on the architecture of the implemented tool. The intended use of the tool is the following: The inputs are a PCRE file and a PCAP file. The PCRE file contains a set of REs describing a protocol or an attack and the PCAP file contains the captured packets. The payloads of the packets are then converted to a format that is used for learning. In the next step, a PA is learned from this file. The REs, stored in the PCRE file, are converted to an NFA. Finally, this NFA and the learned PA are inputs of the

[1] `http://www.numpy.org`
[2] `http://pythonhosted.org/PuLP/`
[3] `http://pythonhosted.org/FAdo/`

approximate reduction or the computation of the probabilistic distance. The implemented prototype tool uses external tools for converting REs to NFAs and for learning of PAs. We briefly describe these tools.

For learning of the PA, which represents the input traffic, the Treba tool is used. Treba is a tool for learning PAs and HMMs [2]. It supports the Alergia algorithm, Baum-Welch training, Viterbi training for HMMs, etc. This tool participated in the competition on learning probabilistic automata and hidden Markov models, called PAutomaC in 2012 (team Hulden) [36]. The learned PAs are stored in the Treba format. To make the learning process more automatic, our implemented tool contains a module for converting payloads of packets to a format used for learning with the Treba tool. The captured packets are loaded from a PCAP file, which can be obtained from the network traffic.

For converting regular expressions (PCREs[4]) to NFAs, the tool Netbench is used. Netbench is a framework for experiments with packet processing algorithms [31]. It also provides algorithms for synthesizing automata into a HW implementation to be used for HW accelerated network traffic filtering. For our requirements, Netbench is extended to provide an export of NFAs to a FA format (a proposed format for representation of NFAs, similar to the Treba format and FAdo format).

The high-level architecture of the implemented system, including external components, is shown in Figure 6.1.

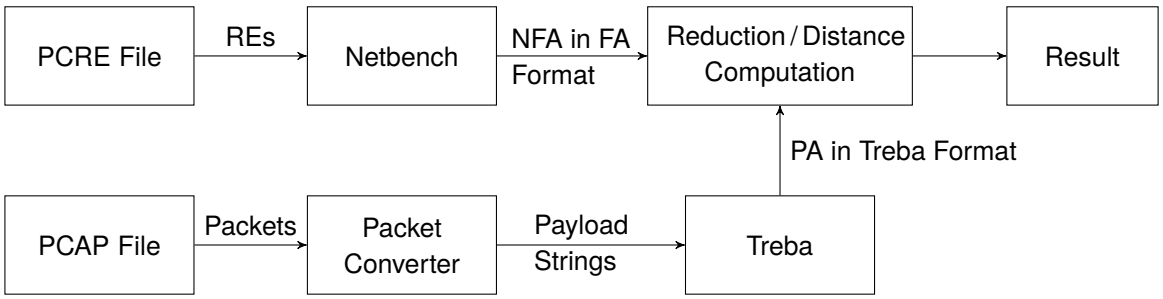


Figure 6.1: The architecture of the implemented system (including external tools Treba and Netbench).

6.2 Basic Classes and Operations

The fundamental building blocks of the implemented tool are classes representing WFAs and NFAs. These classes also offer basic operations needed for computing the distance of input languages and for the approximate reduction. We list these classes and some important operations they provide.

`CoreWFA` A base class for representing WFAs. The WFAs are represented as a list of transitions. The class provides the trim operation, product operation, automata restriction, renaming states, and others. For the purpose of visualization, this class also offers export to the Graphviz[5] DOT language.

[4] Perl Compatible Regular Expressions

[5] `http://www.graphviz.org`

`NFA` A class derived from `CoreWFA` and intended for representing NFAs. It provides disambiguation, backward automata identification, division of an automaton to subautomata, exporting to the FAdo format, and others.

`MatrixWFA` A class derived from `CoreWFA` and intended for matrix-based operations over WFAs. It provides construction of the transition matrix, initial and final vectors, computation of the transition closure, and others. Note that, by the *transition closure*, we mean the computation of $(\mathbf{I} - \mathbf{A}_\Sigma)^{-1}$.

The input NFAs are required in the FA format, and the input PAs are required in the Treba format. The classes `NFAParser` and `WFAParser` thus import such specified automata into the inner representation. Except of these mentioned classes, our tool also contains classes implementing the reduction methods proposed in this work. These classes are described in the following sections.

6.2.1 Computation of the Transition Closure

In this section, we look in more detail on problems related to the computation of the transition closure. As we said in the previous sections and chapters, the computation of the transition closure

$$\sum_{t \geq 0} \mathbf{A}_\Sigma^t = (\mathbf{I} - \mathbf{A}_\Sigma)^{-1} \tag{6.1}$$

is the most time consuming operation in our approach. By default, the matrix inversion is computed numerically by Gaussian elimination. This may, however, cause problems, especially if the matrix has bad properties (e.g., the matrix is ill-conditioned). Therefore, more methods for computing the transition closure are implemented in this tool.

Direct inversion The first of the implemented approaches uses the standard function from the `NumPy` library for computation of the matrix inversion. This approach, we denote as the *direct inversion*.

Iterative multiplying The second implemented approach we denote as the *iterative multiplying*. It is based on the evaluation of the finite sum $\sum_{t=0}^{n} \mathbf{A}_\Sigma^t$ where n is the maximal number of iterations. Clearly, the accuracy of this method depends on the value of n. When we compute the probability (or the weight) of a language L using this method, we actually consider only the probability (or the weight) of strings from L with length at most n.

Hotelling-Bodewig algorithm The last implemented method is referred as *Hotelling-Bodewig algorithm*. The matrix inversion is computed using the following recurrent formula [33]:

$$\mathbf{V}_{n+1} = \mathbf{V}_n(2\mathbf{I} - \mathbf{A}\mathbf{V}_n), \quad n = 0, 1, 2, \ldots \tag{6.2}$$

Here, $\mathbf{V}_0 = \mathbf{I}$, and $\mathbf{A}$ is the matrix whose inverse we want to compute. Except of this simple method there are other possible (and more complicated) ways for an iterative computation of the matrix inversion (e.g., [33]), but they are not considered in this thesis.

The advantage of the direct computation of the matrix inversion is the time complexity. The direct inversion method is faster than both mentioned iterated approximations (this is

caused by the repetitive computations of the matrix multiplication). On the other hand, the direct inversion method can be numerically unstable. An experimental comparison of the rate of convergence of the considered methods in our context is presented in Section 7.2.

6.3 Probabilistic Distance Computation

In this section, we describe some interesting implementation details related to the approximate reduction and the probabilistic distance computation. The computation of the probabilistic distance is implemented according to the algorithm described in Chapter 4. The probabilistic distance is primarily used for measuring the difference between the original and the reduced automaton. We also stated, that for unambiguous automata, the described algorithm computes the distance in polynomial time. However, if we perform the self-loop reduction on an unambiguous automaton, the reduced automaton need not be unambiguous, and therefore, the computation of the distance may be practically infeasible (disambiguation may yield a huge automaton, which is further used for the computation of the transition closure). For this reason, an upper-bound on the probabilistic distance is computed during the reduction as described in Chapter 5. Therefore, if we do not need the exact distance, we can use this estimation.

Apart from the above mentioned approximation, further improvements of the distance computation have been implemented. For instance, if we *a priori* know that $L(A) \subseteq L(A')$, we do not need to compute the product automaton $A \cap A'$ because $L(A) = L(A) \cap L(A')$ (we can take A instead of the product automaton).

6.4 Approximate Reduction

For the approximate reduction of automata, both the pruning and the self-loop reduction introduced in Chapter 5 have been implemented (including all their variants). Basic operations needed for the pruning and the self-loop reduction are provided by the class `CoreReduction`. This class provides operations for computing the values $\theta_{P,A}$ and $\beta_{P,A}$ using the product construction or the approach based on subautomata described in Section 5.4.

6.4.1 Pruning Reduction

The pruning reduction is provided by the class `PruningReduction` (derived from the class `CoreReduction`). This class contains methods for each variant of the pruning reduction.

For solving the relaxed pruning reduction via integer linear programming, the library `PuLP` is used. To recall, in the relaxed k-pruning reduction formulated as the integer linear program, we are looking for a vector $\boldsymbol{x}$ satisfying System 5.13. If this system has more feasible solutions, we want to select the one with the least error. However, there is a small problem with the implementation of the relaxed k-pruning reduction because the `PuLP` library does not allow one to iterate over all feasible solutions. This can be solved by iterative solving of the integer linear programming problem with a dynamic addition of constraints. Consider that we have found some feasible solution $\boldsymbol{x}_1$ with the value $V = \boldsymbol{v}^\top \cdot \boldsymbol{x}_1$ and error $E = \boldsymbol{c}^\top \cdot \boldsymbol{x}_1$ satisfying System 5.13. Then, we add new constraints $\boldsymbol{v}^\top \cdot \boldsymbol{x} \geq V$ and $\boldsymbol{c}^\top \cdot \boldsymbol{x} \leq E$ in order not to obtain a worse solution as the solution found so far in the next iteration. We also add the constraint $\boldsymbol{x}_1^\top \cdot \boldsymbol{x} \leq \boldsymbol{x}_1^\top \cdot \boldsymbol{1} - 1$ to obtain a different solution. After we add these constraints, we solve the modified integer LP problem and if

`PuLP` find some feasible solution, we again add new constraints for the found solution. We repeat this process until some feasible solution is found or we reach the maximum number of iterations.

6.4.2 Self-loop Reduction

The self-loop reduction is provided by the class `SelfLoopReduction` (derived also from `CoreReduction`). The variants of the self-loop reduction are implemented according to Chapter 5. In addition, the following heuristics was used: If addition of self-loops over each symbol to a state does not reduce the size of the automaton, then these self-loops are not included.

6.4.3 State Labels

The computation of the state labels is the most time consuming operation of the whole reduction process (line 1 in Algorithms 6, 7, 8, 9). As we have already said, the complexity is related to disambiguation and the computation of the matrix inverses. If the input automaton is unambiguous, we can use the improved approach described in Section 5.4.

As a further optimization, the implementation of the computation of the state labels uses the structure of the input automaton as well. Since the input NFA is obtained from a union of regular expressions, the obtained structure of the input NFA often looks as shown in Figure 5.5. Therefore, we can divide the whole automaton into subautomata (according to Figure 5.5) and compute the state labels on each subautomaton separately. This can be done because there is no "interaction" between states of the distinct subautomata (when not considering the initial state). Hence, the state labels are computed according to the following steps:

1. Divide the input NFA A into subautomata $\{A_1, A_2, \ldots, A_n\}$.
2. For each subautomaton A_i, $1 \leq i \leq n$, compute the state labels.
3. Union the computed state labels to obtain the state labels of the NFA A (the labels computed in the second step together form the state labels of A).

Moreover, since the subautomata are independent, the second step can run in parallel. The implemented tool also offers a possibility of storing of the computed state labels into a file. These stored values can then be used for repeated reduction attempts with different values of the restriction parameter. This way, the repeated reduction attempts are greatly accelerated since the state labels do not need to be recomputed.

Chapter 7

Experiments

This chapter provides results of experiments performed with the implemented prototype tool for the approximate reduction of automata. The experiments were performed on a PC with Intel Core i5, 3.3×4 GHz, 16 GiB of memory, and running Linux Debian Jessie. The first set of experiments is devoted to learning of probabilistic automata. In the second set, we focus on a comparison of the implemented methods for computing the transition closure. The final set of experiments is then devoted to an evaluation of the proposed automata reduction methods.

7.1 Learning of Probabilistic Automata

For our experience with learning of PAs, we use a sample of network traffic provided by the ANT@FIT group. Together, we obtained several streams of captured packets of size more than 20 GiB. For the learning, the Alergia algorithm implemented in the Treba tool is used. In this experiment, we want to investigate the possibilities of learning of a PA representing the input traffic. As we have already said, the time complexity of the proposed reductions depends on the size of the input PA. Therefore, we want to obtain a suitable PA for further experiments, and determine the influence of the threshold parameter t_0 (see Section 2.4.2 for details) and the number of input packets on the size of the learned PA. In the experiment, the standard Alergia parameter $\alpha = 0.05$ is chosen. The results are shown in Table 7.1.

From the table, we can see that the learning process with the Treba tool is very memory-intensive. Also, it is worth of noticing how much the number of states increases when we change the parameter t_0 to 2. This increase is caused by the nature of the data (sample) used for learning. Most of the captured packets are different from each other. Therefore, the learned PA contains long branches, which are not merged by the Alergia algorithm.

Since the complexity of the probabilistic distance computation and our automata reductions depends on the size of the PA used, we will further use the PA learned from 2000 packets (this PA we denote as P_{2k}).

For the future, an important direction of research is how to improve the process of obtaining the PA characterizing the network traffic.

Table 7.1: The results of the learning of probabilistic automata with the TREBA tool for the parameter $t_0 \in \{0, 1\}$. The column "Packets" denotes the number of used packets for the learning process and the column "States" denotes the size (the number of states) of the learned PA. By "OM", we denote "Out of memory". By the symbol *, we denote that during the learning process, the number of states of the final PA was determined by the TREBA tool, but the learning was not completed because of a lack of memory.

(a) Learning with parameter $t_0 = 1$.

Packets	States	Time
1000	95	3.4s
2000	113	6.4s
8000	225	33.9s
10 000	268	2m 26.6s
20 000	OM	–

(b) Learning with parameter $t_0 = 2$.

Packets	States	Time
1000	455 607*	–
2000	679 410*	–
8000	1 354 569*	–
10 000	1 594 324*	–
20 000	OM	–

7.2 Convergence of the Transition Closure

In Chapter 6, we described three possible ways of computing the transition closure. In the following experiment, we compare the rate of convergence of these methods. In our automata reduction techniques, we use the transition closure for computing the value of $f(\Sigma^*)$. Therefore, we compare here the convergence to this value, depending on the way of computing the closure. The value of $f(\Sigma^*)$ is evaluated on the product of a learned PA and an NFA used in network traffic monitoring. For the experiment, the NFA `info.rules` with 16 states (denoted as A) and the probabilistic automaton P_{2k} learned from 2000 packets is chosen. Then, the transition matrix $\mathbf{A}_\Sigma$, corresponding to the WFA $P' = \mathsf{trim}(P_{2k} \odot A)$ has 1355 rows.

We compute the value of $f_{P'}(\Sigma^*)$ using the direct inversion method, iterative multiplying, and the Hotelling-Bodewig algorithm. Using the direct inversion, we get the value $f_{P'}(\Sigma^*) = 4.16 \times 10^{-19}$ after 10s. The results of the other two methods, which both iteratively improve the precision of the result are shown, for various number of the iterations, in Table 7.2.

As we can see from the table, the computation of the closure via the Hotelling-Bodewig algorithm converges to an exact value $f_{P'}(\Sigma^*)$ computed by the direct inversion much faster than via the iterative multiplying method. Therefore, if the direct inversion cannot be used, the Hotelling-Bodewig algorithm can be more appropriate than the iterative multiplying.

7.3 Automata Reduction

The next experiments are finally devoted to the proposed reduction of automata. For these experiments, we use the probabilistic automaton P_{2k} learned from 2000 packets as described in Section 7.1. In the experiments, we evaluated the algorithms proposed in Chapter 5 on several automata obtained from regular expressions provided to us by the ANT@FIT group.

In the experiments, we, in particular, proceeded as follows: We reduced the automata using the different proposed approaches. For the obtained reduced automata, we computed the probabilistic distance (for bigger automata, we computed only the distance upper-

Table 7.2: A comparison of the rate of convergence of $f_{P'}(\Sigma^*)$ for two iterative methods: the Hotelling-Bodewig algorithm and the iterative multiplying.

(a) Hotelling–Bodewig

Iterations	$f_{P'}(\Sigma^*)$	**Time**
2	0.0	11.4s
5	7.36×10^{-20}	13.1s
10	3.29×10^{-19}	15.9s
50	4.16×10^{-19}	40.1s

(b) Iterative multiplying

Iterations	$f_{P'}(\Sigma^*)$	**Time**
50	8.25×10^{-20}	23.4s
100	1.04×10^{-19}	36.8s
500	2.37×10^{-19}	2m 23s
2000	3.93×10^{-19}	10m 47s
3000	4.10×10^{-19}	15m 42s
5000	4.16×10^{-19}	23m 17s

bound). Moreover, we also determined the error on the sample of traffic provided to us by the ANT@FIT group, which we already used to obtain the PA P_{2k}.

We see the payload of the captured packets as strings. If some packet has no payload, we consider this packet as an empty string ε. The error is then computed as the quotient of the number of misclassified packets and the number of all packets.

When the automata are synthesized into hardware, the input string is accepted, when a final state is reached during the computational steps (the input string may not be completely processed). Therefore, when we compute the error, we also use this "prefix acceptance" (if a string w is accepted in this way by some NFA A, we denote it as $w \in_{pr} L(A)$). The packet p is thus misclassified iff $p \in_{pr} L(A) \oplus p \in_{pr} L(A_r)$ where A is the original NFA, A_r is the reduced NFA, and $\oplus$ is logical nonequivalence.

Let us now present more detailed information about reductions of each considered automaton.

http-bots The first automaton we consider is `http-bots`. The automaton was obtained from the regular expression, which describes a part of the HTTP protocol. The automaton has 8 states only. We performed the k-pruning, and the k-self-loop reduction for various values of k. The results of the reduction are shown in Table 7.3. The error related to the captured traffic (the column "Traffic error") is obtained from a sample of 10^6 packets. The traffic error evaluation took about 10 minutes for each pair of the reduced and the original automaton.

From the table, we can see that, in the case of the self-loop reduction, the computed probabilistic distance is greater than the traffic error, and the difference between the probabilistic distance and the traffic error is quite small. Moreover, observe that we could remove from the original NFA more than half of the states with the traffic error being less than 1 %.

In the case of the k-pruning reduction, the difference between the traffic error and the computed probabilistic distance is bigger. The difference seems to be caused mainly by inaccuracy of the learned PA.

This experimental result can be supported by the following reasoning. In the case of the self-loop reduction, we get that the probabilistic distance between the languages of the original and the reduced automaton is an upper-bound of the traffic error. In addition, the automaton obtained via the self-loop reduction decides on the acceptance of an input string after reading some prefix. This kind of acceptance matches well with the fact that, in our network monitoring application, we can classify some packets after certain prefix of its payload. Also, the hardware accelerated devices usually implement the above mentioned prefix acceptance for a given NFA. Moreover, the self-loop reduction over-approximate the language, which is necessary for some applications (e.g., the error detection). Therefore, the self-loop reduction is probably more suitable for our application and in the further experiments, we focus mainly on the self-loop reduction.

Table 7.3: The reduction of the automaton `http-bots` with 8 states. The figures are shown in Appendix B.

(a) The self-loop reduction

k	States	Traffic error	Probabilistic distance	Figure
0.0	1	0.992	0.999	B.1a
0.2	3	1.69×10^{-3}	7.86×10^{-3}	B.1b
0.5	4	8.90×10^{-4}	3.72×10^{-3}	B.1c
0.6	5	6.00×10^{-6}	6.86×10^{-5}	B.1d
0.7	6	0.0	1.34×10^{-5}	B.1e
1.0	8	0.0	0.0	B.1f

(b) The pruning reduction

k	States	Traffic error	Probabilistic distance	Figure
0.0	1	7.18×10^{-3}	2.00×10^{-8}	B.2a
0.5	4	8.41×10^{-4}	2.73×10^{-13}	B.2b
0.6	5	6.34×10^{-3}	2.00×10^{-8}	B.2c
0.7	6	0.0	0.0	B.1f
1.0	8	0.0	0.0	B.1f

info.rules The second automaton we used for our experiments with the reduction methods is `info.rules`. The automaton has 16 states. It was obtained from the regular expression, which describes authentication messages. We performed the k-self-loop reduction for various values of k. The results of the reduction are shown in Table 7.4. The error related to the captured traffic is obtained from the same sample as in the case of the previous automaton. The evaluation of the traffic error took about 9 hours for each pair of the reduced and the original automaton.

The computed probabilistic distance is again greater than the traffic error. From the table, we can see that, for the k-self-loop reduction with the parameter $k = 0.2$, we obtained

an automaton, which has only 25 % of states of the original automaton, and still the traffic error is less than 1 %.

Table 7.4: The self-loop reduction of the automaton `info.rules` with 16 states. The figures are shown in Appendix B.

k	States	Traffic error	Probabilistic distance	Figure
0.0	1	1.0	1.0	B.3a
0.2	4	8.61×10^{-3}	3.04×10^{-2}	B.3b
0.5	9	0.0	4.04×10^{-10}	B.3c
0.7	12	0.0	1.93×10^{-12}	B.3d
1.0	16	0.0	0.0	B.3e

shellcode.rules The next considered automaton is `shellcode.rules` with 95 states. It was obtained from the regular expression representing several types of shellcodes. We again performed the k-self-loop reduction for various values of k. The results are shown in Table 7.5. The error related to the captured traffic is obtained from a sample of 5×10^5 packets, and the evaluation of the traffic error took about 12 hours for each pair of the reduced and the original automaton.

For each of the reduced automata, we give an upper-bound of the probabilistic distance only. It is because the reduced automaton is no longer an unambiguous automaton. For the computation of the exact distance, we therefore need the disambiguation of the reduced automaton, which would yield a huge automaton.

The computed upper-bound of the probabilistic distance is not greater than the traffic error for the automata reduced with the parameters $k = 0.3$ and $k = 0.5$. This is caused by the inaccuracy of the learned PA. We can again see that we were able to achieve a reduction of 70 % with the traffic error less than 1 %.

Table 7.5: The self-loop reduction of the automaton `shellcode.rules` with 95 states.

k	States	Traffic error	Probabilistic distance
0.0	1	1.0	$\leq$1.0
0.3	29	1.60×10^{-5}	$\leq 1.27 \times 10^{-12}$
0.5	48	1.40×10^{-5}	$\leq 6.41 \times 10^{-19}$
0.7	67	0.0	$\leq 5.29 \times 10^{-25}$
1.0	95	0.0	0.0

chat.rules The last considered automaton is `chat.rules` with 219 states. The automaton was obtained from the regular expression representing some parts of chat protocols (e.g., IRC). We again performed the k-self-loop reduction for various values of k. The results are shown in Table 7.6. Since this automaton is ambiguous, the reduction is more time-demanding (more concretely, the computation of the state labels took about 12 hours). However, this computation is performed only once, independently on the number of the performed reductions. The error related to the captured traffic is obtained from 10^5 packets,

and the evaluation took about 13 hours for each pair of the reduced and the original automaton.

For the same reason as in the case of the previous automaton, we give only an upper-bound of the probabilistic distance for each reduced automaton only. The computed upper-bound of the probabilistic distance is not greater than the traffic error for the automata reduced with the parameter $k = 0.5$. We can again see that we were able to achieve a reduction of 70 % with the traffic error about 3 %.

Table 7.6: The self-loop reduction of the automaton `chat.rules` with 219 states.

k	States	Traffic error	Probabilistic distance
0.0	1	1.0	$\leq$1.0
0.2	47	0.27	$\leq$1.0
0.3	66	3.15×10^{-2}	$\leq 1.02 \times 10^{-1}$
0.5	110	4.00×10^{-5}	$\leq 3.74 \times 10^{-8}$
0.7	154	0.0	$\leq 2.82 \times 10^{-16}$
1.0	219	0.0	0.0

Discussion

In this section, we performed experiments with reductions of automata used in network traffic filtering through the proposed approaches. The results have shown several things: The accuracy of the reduction depends a lot on the learned probabilistic automaton. Currently, we are able to use a restricted subset of the captured packets for the learning only. This could be improved by a packet pre-filtering (briefly discussed in the final chapter). From the first experiments, we also get that the self-loop reduction is probably more suitable for use in network traffic filtering. This conclusion has been confirmed by the people from the ANT@FIT group as well. For some applications in network filtering, it is crucial to make no false negatives (e.g., in attacks detection). Therefore, for this mentioned application, only over-approximate reductions are allowed. Note that, using the over-approximate reductions may cause an increase of the network flow processed by the software resolver. Therefore, for a real deployment, it is necessary to select the restriction parameters with respect to the trade-off between the size of the automaton and the increase of the workload of the software resolver.

The experiments have also shown that the most time-demanding operation is the evaluation of the traffic error based on the captured packets. It is mainly because the test whether a string (payload of a packet) belongs to the language of an NFA is done by a simulation of the NFA (the reduced and the original one) in software. This problem could be solved by a hardware accelerated device for the evaluation of the network error.[1]

Despite all the mentioned problems, the first experiments suggest that the proposed reduction techniques, especially the self-loop reduction, are highly promising for use in network traffic filtering. Indeed, using these techniques, we were able to reduce the considered input automata to less than 30 % of their size with the traffic error less than 3 %.

[1]Such HW was not available at the time of writing this thesis. However, it has already been agreed with the ANT@FIT group that the needed experiments will be performed in the near future.

Chapter 8

Conclusion

The aim of this thesis was to propose techniques for approximate reduction of automata used in network traffic monitoring. Since such approximate reductions do not need to preserve the language, a method for quantification of the incurred error was first proposed. In particular, we proposed a notion of probabilistic distance between languages, which utilizes information about the input network traffic. The information about the network traffic is expressed using probabilistic automata expressing probabilities of different packets. We also gave an algorithm for computing the described distance. In the next part, we proposed two approaches for approximate reduction of automata, including algorithms implementing them in an efficient way. The first one—the pruning reduction—is based on removing branches of an input automaton. The second one—the self-loop reduction—is based on adding self-loops to certain states of an input automaton. The choice of branches to remove or states on which self-loops should be added, is steered by the probabilistic automaton representing the input traffic.

The proposed methods for the reductions were implemented and we performed experiments with the reduction of automata used for network traffic filtering. The reductions were steered by a probabilistic automaton learned by the Alergia algorithm from a captured sequence of packets. For each reduced automaton, we computed the error on the captured traffic. The experiments show that we are able to reduce an input automaton over 70 % of its size with the traffic error less than 3 %, Although, we are not yet able to handle large automata, our approach is highly promising for the future.

The most time-consuming part of the experiments concerning the reductions of automata is the traffic error evaluation. However, this operation could be effectively accelerated in HW. Hence, for further experiments, we would like to exploit this possibility.

In the thesis, we used the Alergia algorithm for learning of the probabilistic automaton representing the network traffic with respect to which the given automaton should be reduced. In the future work, we want to deal with more advanced techniques for learning of probabilistic automata. Currently, the possible ways to obtain a better PA seem to be the following: First, one can use some expert knowledge for a pre-filtration of the captured packets, which would then be used for the learning process. For example, one could think of removing some parts of the packets, using their prefixes only, etc. Another possibility lies in trying to use another algorithms for learning of the probabilistic automata and to compare their accuracy in the context of the reduction framework. We would also like to turn our attention to a special form of learning of the probabilistic automata—namely, learning of a classical NFA in the first step and filling in the probabilities in the second step. Further, one could also think of proposing new learning algorithms specialized for the given context.

Moreover, we would like to explore further ways of reducing the input automata, possibly combining the effect of our approximate reductions with the recently proposed approaches for language-preserving reduction of NFAs.

Bibliography

[1] Rabit and Reduce tool. [online]. Available on: `http://languageinclusion.org/doku.php?id=tools`. [cit. 2017-05-10].

[2] Treba tool. [online]. Available on: `https://code.google.com/archive/p/treba/`. [cit. 2017-05-10].

[3] Abdulla, P. A.; Holík, L.; Kaati, L.; et al.: A Uniform (Bi-)Simulation-Based Framework for Reducing Tree Automata. *Electronic Notes in Theoretical Computer Science*. vol. 251. 2009: pp. 27–48.

[4] Badr, A.; Geffert, V.; Shipman, I.: Hyper-minimizing minimized deterministic finite state automata. *RAIRO - Theoretical Informatics and Applications*. vol. 43, no. 1. 2009: pp. 69–94.

[5] Balle, B.; Panangaden, P.; Precup, D.: A Canonical Form for Weighted Automata and Applications to Approximate Minimization. In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '15. IEEE Computer Society. 2015. pp. 701–712.

[6] Bell, J.: The symmetric difference metric. [online]. Available on: `http://individual.utoronto.ca/jordanbell/notes/symmetric-difference.pdf`. 2015. [cit. 2017-05-10].

[7] Clark, C. R.; Schimmel, D. E.: Scalable Pattern Matching for High Speed Networks. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. FCCM '04. IEEE Computer Society. 2004. pp. 249–257.

[8] Colcombet, T.: Unambiguity in Automata Theory. In *Descriptional Complexity of Formal Systems*, *Lecture Notes in Computer Science*, vol. 9118. Springer International Publishing. 2015. pp. 3–18.

[9] Conforti, M.; Cornuejols, G.; Zambelli, G.: *Integer Programming*. Graduate Texts in Mathematics. Springer International Publishing. first edition. 2014. 456 pp.

[10] Cormen, T. H.; Stein, C.; Rivest, R. L.; et al.: *Introduction to Algorithm*. The MIT Press. third edition. 2010.

[11] Denis, F.; Esposito, Y.: Rational stochastic languages. *CoRR, abs/cs/0602093*. 2006.

[12] Dupont, P.; Denis, F.; Esposito, Y.: Links between probabilistic automata and hidden Markov models: probability distributions, learning models and induction algorithms. *Pattern Recognition*. vol. 38, no. 9. 2005: pp. 1349–1371.

[13] de la Higuera, C.: *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press. 2010. 432 pp.

[14] Hogben, L.: *Handbook of Linear Algebra*. Discrete Mathematics and Its Applications. CRC Press. first edition. 2006.

[15] Holík, L.: *Simulations and Antichains for Efficient Handling of Finite Automata*. PhD. Thesis. FIT VUT v Brně. Brno. 2010.

[16] Horn, R. A.; Johnson, C. R.: *Matrix Analysis*. New York, NY, USA: Cambridge University Press. 1986.

[17] Hromkovič, J.: *Theoretical Computer Science: Introduction to Automata, Compurability, Complexity, Algorithmics, Randomization, Communication, and Cryptography*. Springer-Verlag Berlin Heidelberg. 2004.

[18] Ilie, L.; Navarro, G.; Yu, S.: On NFA Reductions. In *Theory is Forever*, *Lecture Notes in Computer Science*, vol. 3113. Springer Berlin Heidelberg. 2004. pp. 112–124.

[19] Ilie, L.; Yu, S.: Algorithms for Computing Small NFAs. In *Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science*. MFCS '02. Springer-Verlag Berlin Heidelberg. 2002. pp. 328–340.

[20] Jiang, T.; Ravikumar, B.: Minimal NFA Problems are Hard. *SIAM Journal on Computing*. vol. 22, no. 6. 1993: pp. 1117–1141.

[21] Kaštil, J.; Kořenek, J.; Lengál, O.: Methodology for Fast Pattern Matching by Deterministic Finite Automaton with Perfect Hashing. In *Proceedings of 12th Euromicro Conference on Digital System Design*. DSD '09. IEEE Computer Society. 2009. pp. 823–289.

[22] Knuth, D. E.: *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley. third edition. 1997.

[23] Kořenek, J.; Kobierský, P.: Intrusion Detection System Intended for Multigigabit Networks. In *2007 IEEE Design and Diagnostics of Electronic Circuits and Systems*. 2007. pp. 361–364.

[24] Košař, V.; Žádník, M.; Kořenek, J.: NFA Reduction for Regular Expressions Matching Using FPGA. In *2013 International Conference on Field-Programmable Technology*. 2013. pp. 338–341.

[25] Mayr, R.; Clemente, L.: Advanced Automata Minimization. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. ACM. 2013. pp. 63–74.

[26] Meyer, C. D.: *Matrix Analysis and Applied Linear Algebra*. Society for Industrial and Applied Mathematics. 2000.

[27] Mohri, M.: Edit-Distance of Weighted Automata: General Definitions and Algorithms. *International Journal of Foundations of Computer Science*. vol. 14, no. 6. 2003: pp. 957–982.

[28] Mohri, M.: Weighted Automata Algorithms. In *Handbook of Weighted Automata*. Springer Berlin Heidelberg. 2009. pp. 213–254.

[29] Mohri, M.: A Disambiguation Algorithm for Finite Automata and Functional Transducers. In *Implementation and Application of Automata: CIAA 2012*, *Lecture Notes in Computer Science*, vol. 7381. Springer Berlin Heidelberg. 2012. pp. 265–277.

[30] Parker, A. J.; Yancey, K. B.; Yancey, M. P.: Regular Language Distance and Entropy. *CoRR: abs/1602.07715*. 2016.

[31] Puš, V.; Tobola, J.; Košař, V.; et al.: Netbench: Framework for Evaluation of Packet Processing Algorithms. *Symposium On Architecture For Networking And Communications Systems*. 2011: pp. 95–96.

[32] Sokolova, A.; Vink, E. d.: Probabilistic Automata: System Types, Parallel Composition and Comparison. In *Validation of Stochastic Systems*, *Lecture Notes in Computer Science*, vol. 2925. Springer-Verlag Berlin Heidelberg. 2004. pp. 1–43.

[33] Soleymani, F.: On a Fast Iterative Method for Approximate Inverses of Matrices. *Communications of the Korean Mathematical Society*. vol. 28, no. 2. 2013: pp. 407–418.

[34] Vanderbei, R. J.: *Linear Programming: Foundations and Extensions*. Second edition. 1996.

[35] Češka, M.; Vojnar, T.; Smrčka, A.: Teoretická informatika. Lecture notes of FIT BUT. 2014. In Czech.

[36] Verwer, S.; Eyraud, R.; de la Higuera, C.: PAutomaC: a probabilistic automata and hidden Markov models learning competition. *Machine Learning*. vol. 96, no. 1. 2014: pp. 129–154.

Appendices

Appendix A

Proofs of Lemmas and Theorems

In this appendix, we give proofs of the lemmas and theorems that we have not presented in the main text.

A.1 Chapter 4

In this section, we give proofs of lemmas and theorems, which are related to the probabilistic distance computation.

A.1.1 Proofs of Section 4.1

The following lemma and theorem related to SPAs are needed for the probabilistic distance computation.

Lemma 35. *If an SPA* $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ *is well-formed, then every submatrix of* $\mathbf{A}_\Sigma$ *whose rows and columns are given by the nodes of a strongly connected component of* $\mathcal{G}(\mathbf{A}_\Sigma)$ *is substochastic.*

Proof. Suppose that C is a strongly connected component of $\mathcal{G}(\mathbf{A}_\Sigma)$ and $\mathbf{A}_\Sigma[C]$ is the submatrix of $\mathbf{A}_\Sigma$ corresponding to the nodes of C. Since $\mathbf{A}_\Sigma$ is at most stochastic, so is $\mathbf{A}_\Sigma[C]$. There may occur two possibilities:

1. The component C contains a final state q_i, i.e., $\boldsymbol{\alpha}_f[i] > 0$ for some i. Then the row sum of the i-th row needs to be less than 1 (due to Inequality 2.11). Therefore, $\mathbf{A}_\Sigma[C]$ is substochastic.

2. The component C does not contain any final state. Then, since P is well-formed, there exists a state q of the component C with an outgoing transition leading to a state not in C. Thus, the row sum of $\mathbf{A}_\Sigma[C]$ corresponding to the state q is less than 1.

In both cases, we get that $\mathbf{A}_\Sigma[C]$ is a substochastic matrix. □

The following theorem deals with the spectral radius of well-formed semi-probabilistic automata.

Theorem 19. *If an SPA* $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ *is well-formed, then* $\rho(\mathbf{A}_\Sigma) < 1$.

Proof. Since $\mathbf{A}_\Sigma$ is at most a stochastic matrix, it holds that $\rho(\mathbf{A}_\Sigma) \leq 1$ (see [14, Chapter 9.4]). Further, we will show that $\rho(\mathbf{A}_\Sigma) \neq 1$ by contradiction. Suppose that $\rho(\mathbf{A}_\Sigma) = 1$. Then, according to Theorem 1, there exists some access-equivalent class B of $\mathbf{A}_\Sigma$, such that $\rho(\mathbf{A}_\Sigma[B]) = 1$. From the definition of an access-equivalent class, the graph $\mathcal{G}(\mathbf{A}_\Sigma[B])$ is strongly connected, and therefore, according to Theorem 3, the matrix $\mathbf{A}_\Sigma[B]$ is irreducible. Moreover, since P is well-formed, from Lemma 35, we obtain that $\mathbf{A}_\Sigma[B]$ is substochastic. Finally, according to Theorem 4, $\rho(\mathbf{A}_\Sigma[B]) < 1$, which is a contradiction. □

A.1.2 Proofs of Section 4.2

The two technical lemmas stated below are used for computing $f_P(L(A))$ for some UFA A and PA P and hence as a basis for proving Theorem 22.

Lemma 36. *Let $A = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ be a trimmed WFA over Σ. Then, for all $p, q \in Q_A$, there is a $k \in \mathbb{N}_0$, $s \in I_A$, and $f \in F_A$ such that for all $m \in \mathbb{N}$, if $\mathbf{A}_\Sigma^m[p, q] > 0$, then $\mathbf{A}_\Sigma^{m+k}[s, f] > 0$.*

Proof. Let us assume that, for some $m \in \mathbb{N}$ and $p, q \in Q_A$, the inequality $\mathbf{A}_\Sigma^m[p, q] > 0$ is satisfied. This implies the existence of a path in the automaton graph $\mathcal{G}(\mathbf{A}_\Sigma)$ between vertices p and q of length m (written as $p \rightsquigarrow^m q$). Moreover, since WFA A is trimmed, there exist paths $s \rightsquigarrow^{k_1} p$ and $q \rightsquigarrow^{k_2} f$ where $s \in I_A$, $f \in F_A$, and $k_1, k_2 \in \mathbb{N}_0$. Together, we get $s \rightsquigarrow^{k_1+m+k_2} f$, and thus $\mathbf{A}_\Sigma^{m+k}[s, f] > 0$ for $k = k_1 + k_2$ (k is independent of m). □

Lemma 37. *Let $A = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ be a trimmed WFA over Σ. Then,*

$$\lim_{n\to\infty} \boldsymbol{\alpha}_0^\top \mathbf{A}_\Sigma^n \boldsymbol{\alpha}_f = 0 \iff \lim_{n\to\infty} \mathbf{A}_\Sigma^n = \mathbf{0}. \tag{A.1}$$

Proof. The implication from right to left is clear. We will prove the implication from left to right by contradiction. So assume that $\lim_{n\to\infty} \boldsymbol{\alpha}_0^\top \mathbf{A}_\Sigma^n \boldsymbol{\alpha}_f = 0$, and, moreover, we assume an existence of some matrix element at the position (i, j), such that $\lim_{n\to\infty} \mathbf{A}_\Sigma^n[i, j]$ is greater than zero or that the limit does not exist. Moreover, due to $\mathbf{A}_\Sigma$ is a nonnegative matrix, for all $m \in \mathbb{N}$, we have $\mathbf{A}_\Sigma^m[i, j] \geq 0$. This, together with the previous assumption, gives us

$$\limsup_{n\to\infty} \mathbf{A}_\Sigma^n[i, j] > 0. \tag{A.2}$$

Further, from Lemma 36, it follows that there exist $k \in \mathbb{N}_0$, $s \in I_A$, and $f \in F_A$ such that

$$\limsup_{n\to\infty} \mathbf{A}_\Sigma^{n+k}[s, f] = \limsup_{n\to\infty} \mathbf{A}_\Sigma^n[s, f] > 0. \tag{A.3}$$

From the definition of matrix multiplication, we get

$$\boldsymbol{\alpha}_0^\top \mathbf{A}_\Sigma^n \boldsymbol{\alpha}_f = \sum_{l=1}^{m} \sum_{k=1}^{m} \boldsymbol{\alpha}_0^\top[k] \cdot \mathbf{A}_\Sigma^n[k, l] \cdot \boldsymbol{\alpha}_f[l] \geq \boldsymbol{\alpha}_0^\top[s] \cdot \mathbf{A}_\Sigma^n[s, f] \cdot \boldsymbol{\alpha}_f[f] \tag{A.4}$$

for $n \in \mathbb{N}$ and $m = rows(\mathbf{A}_\Sigma)$. Therefore,

$$\limsup_{n\to\infty} \boldsymbol{\alpha}_0^\top \mathbf{A}_\Sigma^n \boldsymbol{\alpha}_f \geq \boldsymbol{\alpha}_0^\top[s] \cdot \left(\limsup_{n\to\infty} \mathbf{A}_\Sigma^n[s, f] \right) \cdot \boldsymbol{\alpha}_f[f] > 0, \tag{A.5}$$

which is a contradiction with the assumption $\lim_{n\to\infty} \boldsymbol{\alpha}_0^\top \mathbf{A}_\Sigma^n \boldsymbol{\alpha}_f = 0$. □

We now finally get to the theorem that together with the previous lemmas allows us to compute $f_P(L)$ for a language L given by a UFA and a PA P. The following theorem is an analogy of Theorem 19.

Theorem 22. *Let P be a PA, A be a UFA, and P' be a WFA $P' = \mathsf{trim}(P \odot A) = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$, then $\rho(\mathbf{A}_\Sigma) < 1$.*

Proof. We can determinize A to obtain a deterministic finite automaton $\hat{A}$. If we intersect P with $\hat{A}$, we obtain a semi-probabilistic automaton, which can be transformed into a well-formed SPA $\hat{P}$, i.e., $\hat{P} = \mathsf{wellFormed}(P \odot \hat{A}) = (\hat{\boldsymbol{\alpha}}_0, \hat{\boldsymbol{\alpha}}_f, \{\hat{\mathbf{A}}_a\}_{a\in\Sigma})$. The functions generated by $\mathsf{weighted}(A)$ and $\mathsf{weighted}(\hat{A})$ are equal, which follows from Lemma 20, and therefore $f_{P'} = f_{\hat{P}}$. Moreover, from Lemma 14 and Theorem 19, we obtain

$$f_{\hat{P}}(\Sigma^*) = \hat{\boldsymbol{\alpha}}_0^\top (\mathbf{I} - \hat{\mathbf{A}})^{-1} \hat{\boldsymbol{\alpha}}_f = f_{P'}(\Sigma^*) = \sum_{t=0}^{\infty} \sum_{w\in\Sigma^t} f_{P'}(w). \tag{A.6}$$

If we let $a_n = \sum_{w\in\Sigma^n} f_{P'}(w)$, according to Equation A.6, the series $\sum_{n=0}^{\infty} a_n$ is convergent ($\sum_{n=0}^{\infty} a_n = \hat{\boldsymbol{\alpha}}_0^\top (\mathbf{I} - \hat{\mathbf{A}})^{-1} \hat{\boldsymbol{\alpha}}_f$). Further, according to Equation 2.15, a_n is also expressible as $a_n = \boldsymbol{\alpha}_0^\top \mathbf{A}_\Sigma^n \boldsymbol{\alpha}_f$. Due to the convergence of the series, we get

$$\lim_{n\to\infty} a_n = \lim_{n\to\infty} \boldsymbol{\alpha}_0^\top \mathbf{A}_\Sigma^n \boldsymbol{\alpha}_f = 0, \tag{A.7}$$

and, according to Lemma 37, we obtain $\lim_{n\to\infty} \mathbf{A}_\Sigma^n = \mathbf{0}$. From Theorem 5, we finally get $\rho(\mathbf{A}_\Sigma) < 1$. □

A.2 Chapter 5

In this section, we give proofs of lemmas and theorems, which are related to the approximate reduction of automata.

A.2.1 Proofs of Section 5.3

The following lemmas are used for an estimation of the error of the self-loop reduction.

Lemma 38. *Let $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ be a trim PA over Σ. Then, for every $w \in \Sigma^*$, we have*

$$f_P(w.\Sigma^*) = \mathsf{weight}_P(w). \tag{A.8}$$

Proof. First, we get

$$\begin{aligned} f_P(w.\Sigma^*) &= \sum_{w'\in\Sigma^*} \boldsymbol{\alpha}_0^\top \mathbf{A}_w \mathbf{A}_{w'} \boldsymbol{\alpha}_f = \boldsymbol{\alpha}_0^\top \mathbf{A}_w \cdot \left(\sum_{w'\in\Sigma^*} \mathbf{A}_{w'} \right) \cdot \boldsymbol{\alpha}_f = \\ &= \boldsymbol{\alpha}_0^\top \mathbf{A}_w (\mathbf{I} - \mathbf{A}_\Sigma)^{-1} \boldsymbol{\alpha}_f \end{aligned} \tag{A.9}$$

The last equality above is due to the equality $\sum_{w'\in\Sigma^*} \mathbf{A}_{w'} = \sum_{t\in\mathbb{N}} \mathbf{A}_\Sigma^t$ and Theorem 19. Further, due to P being a probabilistic automaton, for all $1 \le i \le n$ where $n = rows(\mathbf{A}_\Sigma)$, we get the following from the definition of PA.

$$\begin{aligned} \boldsymbol{\alpha}_f[i] =& 1 - \sum_{j=1}^{n} \mathbf{A}_\Sigma[i,j] = \sum_{j=1}^{n} \mathbf{I}[i,j] - \sum_{j=1}^{n} \mathbf{A}_\Sigma[i,j] = \\ =& \sum_{j=1}^{n} (\mathbf{I}[i,j] - \mathbf{A}_\Sigma[i,j]) = ((\mathbf{I} - \mathbf{A}_\Sigma) \cdot \mathbf{1})[i]. \end{aligned} \tag{A.10}$$

Note that, the multiplication by a vector $\mathbf{1}$ in the last equality only replaces the explicit summation. Thus, we have $\boldsymbol{\alpha}_f = (\mathbf{I} - \mathbf{A}_\Sigma) \cdot \mathbf{1}$. From Theorems 5 and 19, we obtain that the matrix $(\mathbf{I} - \mathbf{A}_\Sigma)$ is invertible. Therefore, we can multiply both sides by $(\mathbf{I} - \mathbf{A}_\Sigma)^{-1}$, and we get $(\mathbf{I} - \mathbf{A}_\Sigma)^{-1}\boldsymbol{\alpha}_f = \mathbf{1}$. Finally, by a substitution to Equality A.9, we have

$$f_P(w.\Sigma^*) = \boldsymbol{\alpha}_0^\top \mathbf{A}_w (\mathbf{I} - \mathbf{A}_\Sigma)^{-1} \boldsymbol{\alpha}_f = \boldsymbol{\alpha}_0^\top \mathbf{A}_w \cdot \mathbf{1} = \mathsf{weight}_P(w), \tag{A.11}$$

which concludes the proof. □

Lemma 28. *Let P be a trim PA over Σ, and $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Then, for all $q \in Q$ we have*

$$f_P(L_A^{-1}(q).\Sigma^*) \leq \beta_{P,A}(q). \tag{A.12}$$

Proof. We start with the following reasoning

$$\begin{aligned} f_P(L_A^{-1}(q).\Sigma^*) &= \sum_{w \in L_A^{-1}(q).\Sigma^*} f_P(w) \\ &\leq \sum_{w \in L_A^{-1}(q)} f_P(w.\Sigma^*). \end{aligned} \tag{A.13}$$

The previous inequality holds because there can be strings $w_1, w_2 \in L_A^{-1}(q)$ such that $w_1.\Sigma^* \cap w_2.\Sigma^* \neq \emptyset$. Finally, from Lemma 38 and Inequality A.13, we obtain

$$f_P(L_A^{-1}(q).\Sigma^*) \leq \sum_{w \in L_A^{-1}(q)} f_P(w.\Sigma^*) = \sum_{w \in L_A^{-1}(q)} \mathsf{weight}_P(w) = \beta_{P,A}(q), \tag{A.14}$$

which concludes the proof. □

A.2.2 Proofs of Section 5.4

The below lemma clarifies the relation between the multiplication of the transition matrices of NFA, PA, and their product. This lemma is then used for a more efficient way of computing the values $\theta_{P,A}$ and $\beta_{P,A}$.

Lemma 39. *Let $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA, let $\{\mathbf{T}_a\}a_\Sigma$ be its transition matrices encoding its transitions (i.e., transition matrices of $\mathsf{weighted}(A)$), and let $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a \in \Sigma})$ be a PA over Σ. Further, let B be a WFA over Σ given as $B = \mathsf{trim}(P \odot A) = (\boldsymbol{\beta}_0, \boldsymbol{\beta}_f, \{\mathbf{B}_a\}_{a \in \Sigma})$. Then, for each $(q_1, q_2), (r_1, r_2) \in Q_B$ and for each $x \in \Sigma^*$, we have*

$$\mathbf{A}_x[q_1, r_1] \cdot \mathbf{T}_x[q_2, r_2] = \mathbf{B}_x[(q_1, q_2), (r_1, r_2)]. \tag{A.15}$$

Proof. We prove this statement by induction on the length of x. For $x = \varepsilon$, we have

$$\mathbf{I}[q_1, r_1] \cdot \mathbf{I}[q_2, r_2] = \mathbf{I}[(q_1, q_2), (r_1, r_2)]. \tag{A.16}$$

If $q_1 = r_1$ and $q_2 = r_2$, then both sides are equal to 1, otherwise they are equal to 0. Therefore, the equation is valid.

For $x = a \in \Sigma$, according to the 6th line of Algorithm 1, we obtain

$$\mathbf{A}_a[q_1, r_1] \cdot \mathbf{T}_a[q_2, r_2] = \mathbf{B}_a[(q_1, q_2), (r_1, r_2)]. \tag{A.17}$$

For the inductive case, we assume that the equality holds for each $(q_1, q_2), (r_1, r_2) \in Q_B$, and a string x of length at most n. We prove that the equality is satisfied also for a string $x.a$ of length at most $n+1$ where $a \in \Sigma$:

$$\begin{aligned}
&\mathbf{A}_{x.a}[q_1, r_1] \cdot \mathbf{T}_{x.a}[q_2, r_2] = \\
&\quad = \left(\sum_{k \in Q_P} \mathbf{A}_x[q_1, k] \cdot \mathbf{A}_a[k, r_1] \right) \cdot \left(\sum_{l \in Q} \mathbf{T}_x[q_2, l] \cdot \mathbf{T}_a[l, r_2] \right) = \\
&\quad = \sum_{k \in Q_P} \left(\mathbf{A}_x[q_1, k] \cdot \mathbf{A}_a[k, r_1] \cdot \sum_{l \in Q} \mathbf{T}_x[q_2, l] \cdot \mathbf{T}_a[l, r_2] \right) = \\
&\quad = \sum_{k \in Q_P} \sum_{l \in Q} \mathbf{A}_x[q_1, k] \cdot \mathbf{A}_a[k, r_1] \cdot \mathbf{T}_x[q_2, l] \cdot \mathbf{T}_a[l, r_2] = \\
&\quad = \sum_{(k,l) \in Q_P \times Q} \mathbf{A}_x[q_1, k] \cdot \mathbf{A}_a[k, r_1] \cdot \mathbf{T}_x[q_2, l] \cdot \mathbf{T}_a[l, r_2] = \\
&\quad = \sum_{(k,l) \in Q_B} \mathbf{A}_x[q_1, k] \cdot \mathbf{A}_a[k, r_1] \cdot \mathbf{T}_x[q_2, l] \cdot \mathbf{T}_a[l, r_2] + \\
&\qquad + \sum_{(k,l) \in (Q_P \times Q) \setminus Q_B} \mathbf{A}_x[q_1, k] \cdot \mathbf{A}_a[k, r_1] \cdot \mathbf{T}_x[q_2, l] \cdot \mathbf{T}_a[l, r_2].
\end{aligned} \tag{A.18}$$

Now, we look at the second sum of the last expression. Since $(k, l) \notin Q_B$, it means that the state (k, l) was either non-accessible or non-coaccessible in $P \odot A$.

1. If the state (k, l) is non-accessible in $P \odot A$, then there do not exist both a path over x from the state q_1 to the state k in P and a path over x from the state q_2 to the state l in A. If both paths existed, the state (k, l) would be accessible in B because (q_1, q_2) is accessible in B. Therefore, we have $\mathbf{A}_x[q_1, k] \cdot \mathbf{T}_x[q_2, l] = 0$.

2. If the state (k, l) is non-coaccessible (but accessible) in $P \odot A$, then there do not exist both a transition over a from the state k to the state r_1 in P and transition over a from the state l to the state r_2 in A. If both these transitions existed, the state (k, l) would be coaccessible because there would be a path in $P \odot A$ from (k, l) to (r_1, r_2) where the state (r_1, r_2) is coaccessible due to the assumption of the inductive case. We thus have $\mathbf{A}_a[k, r_1] \cdot \mathbf{T}_a[l, r_2] = 0$.

Together, we obtain

$$\sum_{(k,l) \in (Q_P \times Q) \setminus Q_B} \mathbf{A}_x[q_1, k] \cdot \mathbf{A}_a[k, r_1] \cdot \mathbf{T}_x[q_2, l] \cdot \mathbf{T}_a[l, r_2] = 0. \tag{A.19}$$

If $(k, l) \in Q_B$, according to the induction hypothesis, we have

$$\mathbf{A}_x[q_1, k] \cdot \mathbf{T}_x[q_2, l] = \mathbf{B}_x[(q_1, q_2), (k, l)], \tag{A.20}$$

and also

$$\mathbf{A}_a[k, r_1] \cdot \mathbf{T}_a[l, r_2] = \mathbf{B}_a[(k, l), (r_1, r_2)]. \tag{A.21}$$

By substitution, we obtain

$$\sum_{(k,l) \in Q_B} \mathbf{B}_x[(q_1, q_2), (k, l)] \cdot \mathbf{B}_a[(k, l), (r_1, r_2)] = \mathbf{B}_{x.a}[(q_1, q_2), (r_1, r_2)], \tag{A.22}$$

which concludes the proof. □

The following theorem allows us to compute the value of $\theta_{P,A}(q)$ for an NFA A, a PA P, and any $q \in F_A$ in a more efficient way. We are able to compute the $\theta_{P,A}$ values directly from the product $P \odot A$.

Theorem 32. *Let $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ be a trimmed PA over Σ, let $A = (Q, \Sigma, \delta, q_0, F)$ be a trimmed UFA, and let $\{\mathbf{T}_a\}_{a\in\Sigma}$ be the transition matrices of* $\mathsf{weighted}(A)$*. Further, let P' be a WFA given as $P' = \mathsf{trim}(P \odot A) = (\boldsymbol{\beta}_0, \boldsymbol{\beta}_f, \{\mathbf{B}_a\}_{a\in\Sigma})$. Then, for each $q \in F$, we have*

$$\theta_{P,A}(q) = \boldsymbol{\beta}_0^\top \cdot \boldsymbol{\Pi}_{P'} \cdot \boldsymbol{\gamma}_q^{\boldsymbol{\alpha}_f} \tag{A.23}$$

where $\boldsymbol{\Pi}_{P'} = (\mathbf{I} - \mathbf{B}_\Sigma)^{-1} = \sum_{t\geq 0} \mathbf{B}_\Sigma^t$, and, for each $(p', q') \in Q_{P'}$,

$$\boldsymbol{\gamma}_q^{\boldsymbol{\alpha}_f}[(p', q')] = \begin{cases} \boldsymbol{\alpha}_f[p'] & \text{if } q' = q, \\ 0 & \text{otherwise.} \end{cases} \tag{A.24}$$

Proof. First, from Theorem 22 and Lemma 14, it follows that $(\mathbf{I} - \mathbf{B}_\Sigma)^{-1} = \sum_{t\geq 0} \mathbf{B}_\Sigma^t$. Further, from the associativity of matrix multiplication, we get

$$\boldsymbol{\alpha}_0^\top \cdot \mathbf{A}_x \cdot \boldsymbol{\alpha}_f = (\boldsymbol{\alpha}_0^\top \cdot \mathbf{A}_x) \cdot \boldsymbol{\alpha}_f = \sum_{l\in Q_P} (\boldsymbol{\alpha}_0^\top \cdot \mathbf{A}_x)[l] \cdot \boldsymbol{\alpha}_f[l]. \tag{A.25}$$

Similarly, for $(\boldsymbol{\alpha}_0^\top \cdot \mathbf{A}_x)[l]$, we get

$$(\boldsymbol{\alpha}_0^\top \cdot \mathbf{A}_x)[l] = \sum_{k\in Q_P} \boldsymbol{\alpha}_0^\top[k] \cdot \mathbf{A}_x[k, l]. \tag{A.26}$$

If $\boldsymbol{\alpha}_0^\top[k] = 0$, then the whole product is zero. We can thus sum only over the states having the corresponding values nonzero in $\boldsymbol{\alpha}_0^\top$ (i.e., states from the set of initial states I_P). Now, we focus on the computation of $\theta_{P,A}(q)$ for arbitrary $q \in F$. We start with the definition of $f_P(L)$:

$$\begin{aligned} \theta_{P,A}(q) &= \sum_{x\in L_A^{-1}(q)} f_P(x) = \sum_{x\in L_A^{-1}(q)} \boldsymbol{\alpha}_0^\top \cdot \mathbf{A}_x \cdot \boldsymbol{\alpha}_f = && \text{(A.27)} \\ &= \sum_{x\in L_A^{-1}(q)} \sum_{l\in Q_P} \left(\sum_{k\in I_P} \boldsymbol{\alpha}_0^\top[k] \cdot \mathbf{A}_x[k, l] \right) \cdot \boldsymbol{\alpha}_f[l] = && \text{(A.28)} \\ &= \sum_{x\in L_A^{-1}(q)} \sum_{k\in I_P} \sum_{l\in Q_P} \boldsymbol{\alpha}_0^\top[k] \cdot \mathbf{A}_x[k, l] \cdot \boldsymbol{\alpha}_f[l]. && \text{(A.29)} \end{aligned}$$

Since A is a trimmed unambiguous automaton, for each state $p \in Q$, it holds that there exists at most one path from q_0 to p labeled by arbitrary $x \in \Sigma^*$. Indeed, if there were, for some state p and some string x, two distinct paths from q_0 to p, we could extend these paths from p to some final state, which would yield two distinct accepting computations on x in A. This would be a contradiction because A is unambiguous. Therefore, we have $\mathbf{T}_x[q_0, p] = 1$, if $x \in L_A^{-1}(p)$, and $\mathbf{T}_x[q_0, p] = 0$ otherwise. Therefore, we get

$$\begin{aligned} \theta_{P,A}(q) &= \sum_{x\in L_A^{-1}(q)} \sum_{k\in I_P} \sum_{l\in Q_P} \boldsymbol{\alpha}_0^\top[k] \cdot \mathbf{A}_x[k, l] \cdot \boldsymbol{\alpha}_f[l] = && \text{(A.30)} \\ &= \sum_{x\in\Sigma^*} \sum_{k\in I_P} \sum_{l\in Q_P} \boldsymbol{\alpha}_0^\top[k] \cdot \mathbf{A}_x[k, l] \cdot \mathbf{T}_x[q_0, q] \cdot \boldsymbol{\alpha}_f[l]. && \text{(A.31)} \end{aligned}$$

Further, the expression $\boldsymbol{\alpha}_0^\top[k] \cdot \mathbf{A}_x[k,l] \cdot \mathbf{T}_x[q_0,q] \cdot \boldsymbol{\alpha}_f[l]$ is greater than zero only if there exists a path in P from $k \in Q_P$ to $l \in F_P$ labeled by x, and a path in A from q_0 to $q \in F$ labeled by x. Thus, there exists also a path in $P \odot A$ from the initial state (k, q_0) to the final state (l, q). Since these states are both accessible and co-accessible, they are included into the trimmed automaton P'. Therefore, we can narrow the sum as follows:

$$\theta_{P,A}(q) = \sum_{x \in \Sigma^*} \sum_{(k,q_0) \in I_{P'}} \sum_{(l,q) \in Q_{p'}} \boldsymbol{\alpha}_0^\top[k] \cdot \mathbf{A}_x[k,l] \cdot \mathbf{T}_x[q_0,q] \cdot \boldsymbol{\alpha}_f[l]. \tag{A.32}$$

Moreover, according to Lemma 39, the equality $\mathbf{A}_x[k,l] \cdot \mathbf{T}_x[q_0,q] = \mathbf{B}_x[(k,q_0),(l,q)]$ is satisfied. Then, from Equation A.32, we obtain

$$\theta_{P,A}(q) = \sum_{x \in \Sigma^*} \sum_{(k,q_0) \in I_{P'}} \sum_{(l,q) \in Q_{p'}} \boldsymbol{\alpha}_0^\top[k] \cdot \mathbf{B}_x[(k,q_0),(l,q)] \cdot \boldsymbol{\alpha}_f[l] = \tag{A.33}$$

$$= \sum_{x \in \Sigma^*} \boldsymbol{\beta}_0^\top \cdot \mathbf{B}_x \cdot \boldsymbol{\gamma}_q^{\boldsymbol{\alpha}_f} = \tag{A.34}$$

$$= \boldsymbol{\beta}_0^\top \cdot \boldsymbol{\Pi}_{P'} \cdot \boldsymbol{\gamma}_q^{\boldsymbol{\alpha}_f}, \tag{A.35}$$

which concludes the proof. □

The below lemma gives us a way of computing $\mathsf{weight}_P(L(A))$ for a UFA A and a PA P whose all states are final. The way of computing the weights is similar to computing $f_P(L(A))$, the formulas differ in the form of the final vector only.

Lemma 40. *Let $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a \in \Sigma})$ be a PA having all states final, A be a UFA, and $P' = \mathsf{trim}(P \odot A) = (\boldsymbol{\beta}_0, \boldsymbol{\beta}_f, \{\mathbf{B}_a\}_{a \in \Sigma})$ be a WFA. Then,*

$$\mathsf{weight}_P(L(A)) = \boldsymbol{\beta}_0^\top \cdot (\mathbf{I} - \mathbf{B}_\Sigma)^{-1} \cdot \boldsymbol{\gamma}_f^{\mathbf{1}} \tag{A.36}$$

where, for each $q \in Q_{P'}$, the vector $\boldsymbol{\gamma}_f^{\mathbf{1}}$ is given as

$$\boldsymbol{\gamma}_f^{\mathbf{1}}[q] = \begin{cases} 1 & \text{if } \boldsymbol{\beta}_f[q] > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{A.37}$$

Proof. Consider the WFA P_α obtained from P by setting the accepting probabilities of every state to 1, i.e., $P_\alpha = (\boldsymbol{\alpha}_0, \mathbf{1}, \{\mathbf{A}_a\}_{a \in \Sigma})$. Further, consider the WFA $P_{\beta'}$ given as $P_{\beta'} = \mathsf{trim}(P_\alpha \cap \mathsf{weighted}(A)) = (\boldsymbol{\beta}'_0, \boldsymbol{\beta}'_f, \{\mathbf{B}'_a\}_{a \in \Sigma})$. Since P and P_α differ only in accepting probabilities (but both have the same set of final states), the WFAs $P_{\beta'}$ and P' also differ only in accepting probabilities. Therefore, $\boldsymbol{\beta}'_0 = \boldsymbol{\beta}_0$ and $\mathbf{B}'_a = \mathbf{B}_a$ for all $a \in \Sigma$. Also, according to the definition of the product construction, we have $\boldsymbol{\beta}'_f = \boldsymbol{\gamma}_f^{\mathbf{1}}$.

Since P is a PA where all states are final, we have for every $w \in \Sigma^*$

$$\mathsf{weight}_P(w) = f_{P_\alpha}(w). \tag{A.38}$$

Further, from the definition of the product of WFAs and from Lemma 20, we obtain that, for all $w \in L(A)$, we have $f_{P_\alpha}(w) = f_{P_{\beta'}}(w)$, and, for all $w \notin L(A)$, we have $f_{P_{\beta'}}(w) = 0$. Together, we get

$$\mathsf{weight}_P(L(A)) = f_{P_\alpha}(L(A)) = f_{P_{\beta'}}(\Sigma^*). \tag{A.39}$$

Moreover, from Theorem 22, we get that $\rho(\mathbf{B}_\Sigma) < 1$, and therefore, from Lemma 14, we get

$$f_{P_{\beta'}}(\Sigma^*) = \boldsymbol{\beta}_0^\top \cdot (\mathbf{I} - \mathbf{B}_\Sigma)^{-1} \cdot \boldsymbol{\gamma}_f^{\mathbf{1}}, \tag{A.40}$$

which concludes the proof. □

The following theorem allows us to compute $\beta_{P,A}(r)$ for NFA A, PA P, and all $r \in Q_A$ in a more efficient way. However, we put an additional restriction $L(\mathsf{supp}(P)) = \Sigma^*$ on P. Then, the values $\beta_{P,A}(r)$ we are able to compute directly from the product $P \odot A$ (this theorem is analogy to Theorem 32).

Theorem 34. *Let $P = (\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_f, \{\mathbf{A}_a\}_{a\in\Sigma})$ be a trimmed DPA over Σ with just one initial state s, i.e., $I_P = \{s\}$, such that $L(\mathsf{supp}(P)) = \Sigma^*$. Further, let $A = (Q, \Sigma, \delta, q_0, F)$ be a trimmed UFA, and let $\{\mathbf{T}_a\}_{a\in\Sigma}$ be the transition matrices of $\mathsf{weighted}(A)$. We also consider WFA P' given as $P' = \mathsf{trim}(P \odot A) = (\boldsymbol{\beta}_0, \boldsymbol{\beta}_f, \{\mathbf{B}_a\}_{a\in\Sigma})$. Then, for all $r \in Q$, we have*

$$\beta_{P,A}(r) = \boldsymbol{\beta}_0^\top \cdot \boldsymbol{\Pi}_{P'} \cdot \boldsymbol{\gamma}_r^{\mathbf{1}} \tag{A.41}$$

where $\boldsymbol{\Pi}_{P'} = (\mathbf{I} - \mathbf{B}_\Sigma)^{-1} = \sum_{t\geq 0} \mathbf{B}_\Sigma^t$, and, for each $(q', r') \in Q_{P'}$,

$$\boldsymbol{\gamma}_r^{\mathbf{1}}[(q', r')] = \begin{cases} 1 & \text{if } r' = r, \\ 0 & \text{otherwise.} \end{cases} \tag{A.42}$$

Proof. First, from Theorem 22 and Lemma 14 really follows that $(\mathbf{I} - \mathbf{B}_\Sigma)^{-1} = \sum_{t\geq 0} \mathbf{B}_\Sigma^t$. Now, we prove the following equality

$$\forall r \in Q : \bigcup_{(q,r)\in Q_{P'}} L_{P'}^{-1}((q, r)) = L_A^{-1}(r). \tag{A.43}$$

Intuitively, if $w \in L_A^{-1}(r)$ then there exists some $q \in Q_P$, such that $w \in L_{P'}^{-1}((q, r))$, and vice versa (since $L(\mathsf{supp}(P)) = \Sigma^*$). Formally, let us first focus on the inclusion $M = \bigcup_{(q,r)\in Q_{P'}} L_{P'}^{-1}((q, r)) \subseteq L_A^{-1}(r)$. Consider some state $r \in Q$, and some string $w \in M$. If there is a path in P' from (s, q_0) to some (q, r) where $q \in Q_P$, labeled by w, there is also a path in A from q_0 to r labeled by w. Therefore, $w \in L_A^{-1}(r)$.

Now, we look at the reverse inclusion. Consider some $w \in L_A^{-1}(r)$. Since $L(\mathsf{supp}(P)) = \Sigma^*$, there exists some state (q, r) where $q \in Q_P$ in the product automaton, such that $w \in L_{P'}^{-1}((q, r))$. But we still have to prove that $(q, r) \in Q_{P'}$ (i.e., that the state (q, r) is not removed during the trimming). We prove this using the following reasoning: Due to A being a trimmed UFA, there exists some state $f \in F$ such that we reach the state f from r in A by some string w'. Also, since $\mathsf{supp}(P)$ is a DFA, and $L(\mathsf{supp}(\mathsf{P})) = \Sigma^*$, the support automaton accepts the string $w.w'$. Therefore, there exists a path from q to some $f' \in F_P$ in P labeled by w'. And thus $(q, r) \in Q_{P'}$.

In the next step, we prove that for all $(q_1, r), (q_2, r) \in Q_{P'}$, $q_1 \neq q_2$, we have $L_{P'}^{-1}((q_1, r)) \cap L_{P'}^{-1}((q_2, r)) = \emptyset$. We prove this by a contradiction. Assume that there exists some $w \in \Sigma^*$ and states $(q_1, r), (q_2, r) \in Q_{P'}$, $q_1 \neq q_2$, such that $w \in L_{P'}^{-1}((q_1, r)) \cap L_{P'}^{-1}((q_2, r))$. Therefore, $w \in L_P^{-1}(q_1)$, and $w \in L_P^{-1}(q_2)$, which is a contradiction, because $\mathsf{supp}(P)$ is a DFA.

In the last step we prove the main equality. We start with the definition of the $\beta_{P,A}$ function for some $r \in Q$:

$$\beta_{P,A}(r) = \mathsf{weight}_P(L_A^{-1}(r)) = \mathsf{weight}_P\left(\bigcup_{(q,r)\in Q_{P'}} L_{P'}^{-1}((q, r))\right). \tag{A.44}$$

Because the backward languages in the union are disjoint (as shown above), we obtain the following:

$$\beta_{P,A}(r) = \mathsf{weight}_P(L_A^{-1}(r)) = \sum_{(q,r)\in Q_{P'}} \mathsf{weight}_P(L_{P'}^{-1}((q, r))). \tag{A.45}$$

Now, we focus on computing $\mathsf{weight}_P(L_{P'}^{-1}((q,r)))$. Again, we start with the definition of weight:

$$\mathsf{weight}_P(L_{P'}^{-1}((q,r))) = \sum_{x \in L_{P'}^{-1}((q,r))} \boldsymbol{\alpha}_0^\top \cdot \mathbf{A}_x \cdot \mathbf{1}. \tag{A.46}$$

Since s is the only initial state of P, and after we read x we reach q in P, we have

$$\sum_{x \in L_{P'}^{-1}((q,r))} \boldsymbol{\alpha}_0^\top \cdot \mathbf{A}_x \cdot \mathbf{1} = \sum_{x \in L_{P'}^{-1}((q,r))} \boldsymbol{\alpha}_0^\top[s] \cdot \mathbf{A}_x[s,q]. \tag{A.47}$$

Further, since A is unambiguous, there exists only one path from q_0 to r labeled by x. Therefore, if $x \in L_A^{-1}(r)$ then $\mathbf{T}_x[q_0,r] = 1$, and if $x \notin L_A^{-1}(r)$ then $\mathbf{T}_x[q_0,r] = 0$. And thus

$$\begin{aligned}\sum_{x \in L_{P'}^{-1}((q,r))} \boldsymbol{\alpha}_0^\top[s] \cdot \mathbf{A}_x[s,q] &= \sum_{x \in L_{P'}^{-1}((q,r))} \boldsymbol{\alpha}_0^\top[s] \cdot \mathbf{A}_x[s,q] \cdot \mathbf{T}_x[q_0,r] = \\ &= \sum_{x \in \Sigma^*} \boldsymbol{\alpha}_0^\top[s] \cdot \mathbf{A}_x[s,q] \cdot \mathbf{T}_x[q_0,r].\end{aligned} \tag{A.48}$$

We also know that (s,q_0) and (q,r) are states of P'. Therefore, from Lemma 39, we get

$$\sum_{x \in \Sigma^*} \boldsymbol{\alpha}_0^\top[s] \cdot \mathbf{A}_x[s,q] \cdot \mathbf{T}_x[q_0,r] = \sum_{x \in \Sigma^*} \boldsymbol{\alpha}_0^\top[s] \cdot \mathbf{B}_x[(s,q_0),(q,r)]. \tag{A.49}$$

Finally,

$$\sum_{x \in \Sigma^*} \boldsymbol{\alpha}_0^\top[s] \cdot \mathbf{B}_x[(s,q_0),(q,r)] = \sum_{x \in \Sigma^*} \boldsymbol{\beta}_0^\top \cdot \mathbf{B}_x \cdot \boldsymbol{\gamma}^{\mathbf{1}}_{(q,r)} = \boldsymbol{\beta}_0^\top \cdot \boldsymbol{\Pi}_{P'} \cdot \boldsymbol{\gamma}^{\mathbf{1}}_{(q,r)} \tag{A.50}$$

where for each $(q',r') \in Q_{P'}$

$$\boldsymbol{\gamma}^{\mathbf{1}}_{(q,r)}[(q',r')] = \begin{cases} 1 & \text{if } (q',r') = (q,r), \\ 0 & \text{otherwise.} \end{cases} \tag{A.51}$$

Now, if we substitute Equality A.50 into Equality A.45, we get

$$\beta_{P,A}(r) = \sum_{(q,r) \in Q_{P'}} \boldsymbol{\beta}_0^\top \cdot \boldsymbol{\Pi}_{P'} \cdot \boldsymbol{\gamma}^{\mathbf{1}}_{(q,r)} = \boldsymbol{\beta}_0^\top \cdot \boldsymbol{\Pi}_{P'} \cdot \boldsymbol{\gamma}^{\mathbf{1}}_r. \tag{A.52}$$

□

Appendix B

Reduced Automata

In this appendix, we give examples of the reduced automata used in network traffic filtering. For clarity, multiple transitions between two states over different symbols are replaced by a single transition labeled with a set of symbols. Note that, non-printable symbols are denoted by their hexadecimal numbers.

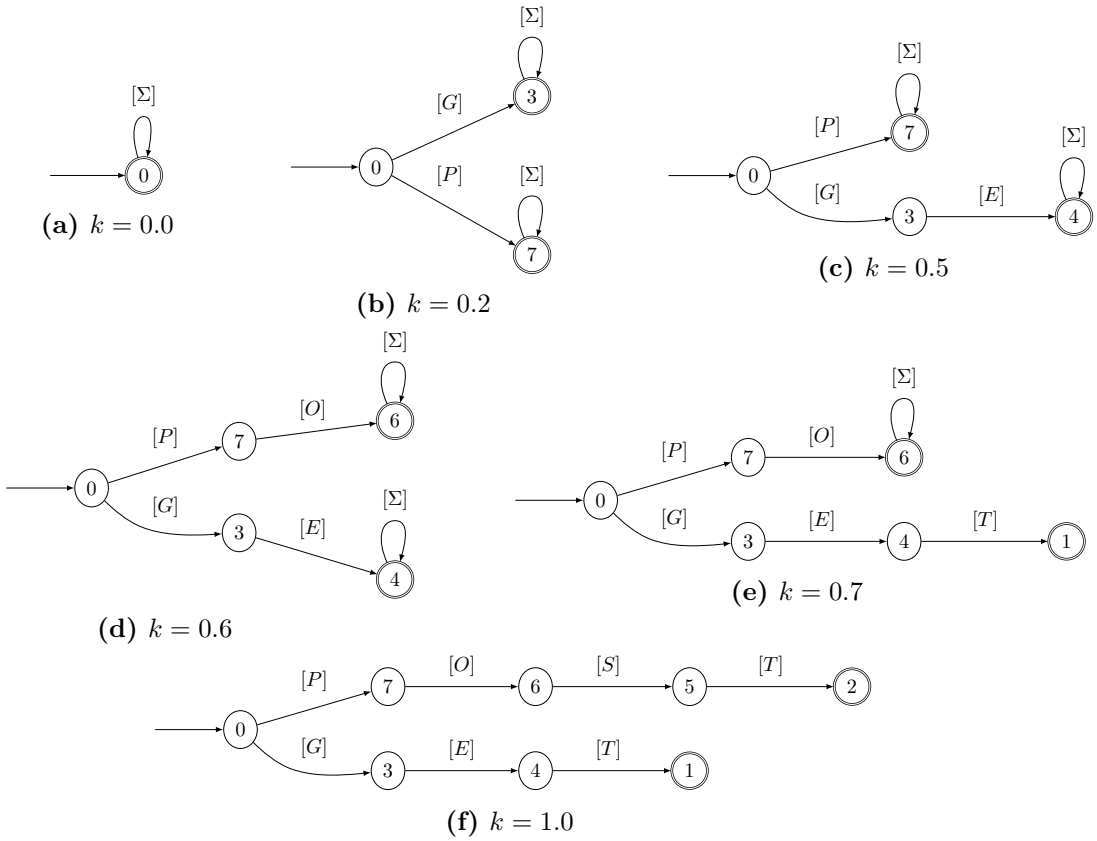
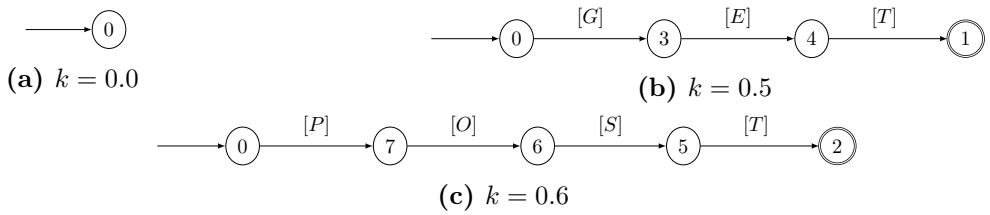


Figure B.1: The k-self-loop reduction of the automaton `http-bots`.

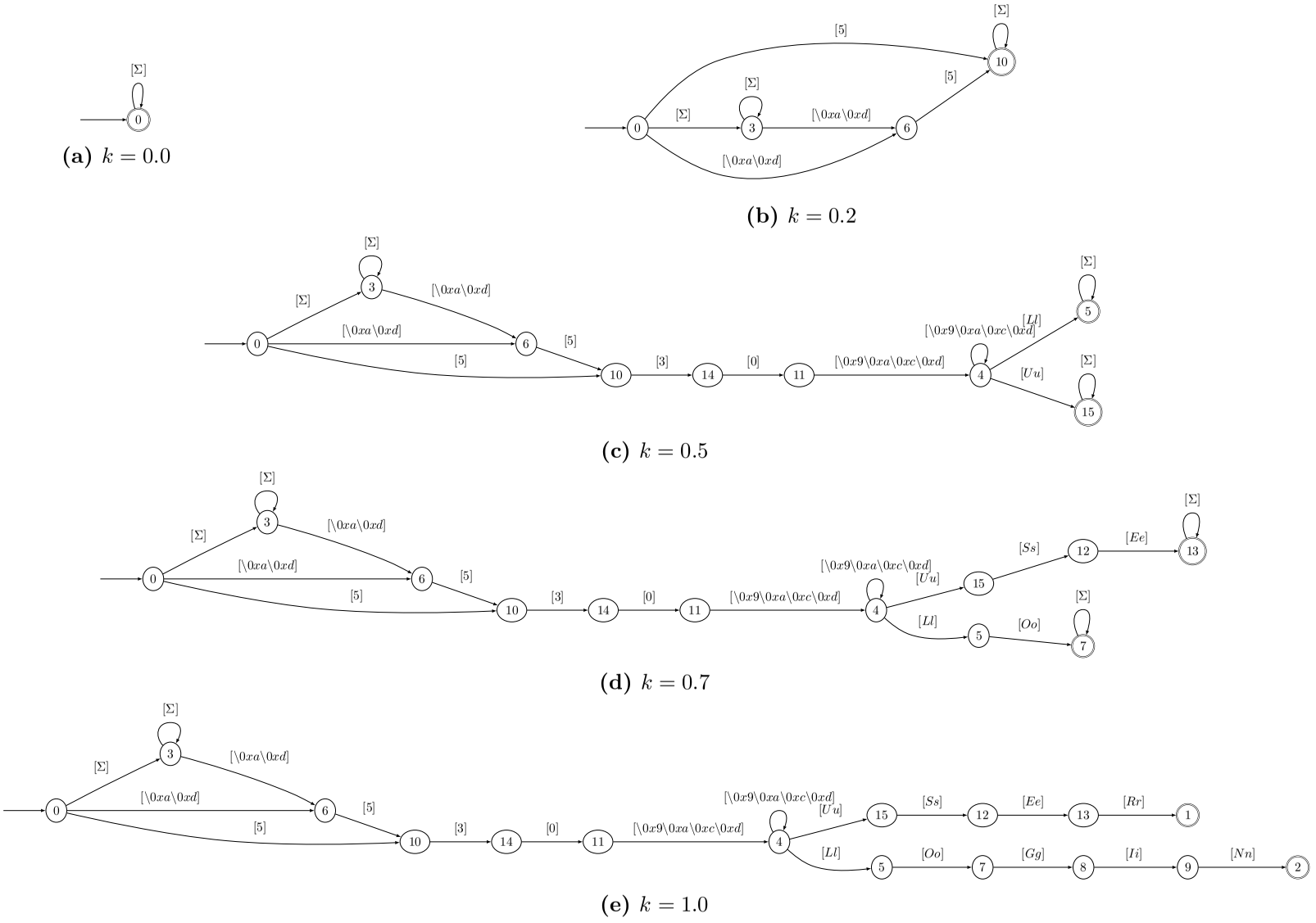


(a) $k = 0.0$

(b) $k = 0.5$

(c) $k = 0.6$

Figure B.2: The k-pruning reduction of the automaton `http-bots`.



(a) $k = 0.0$

(b) $k = 0.2$

(c) $k = 0.5$

(d) $k = 0.7$

(e) $k = 1.0$

Figure B.3: The k-self-loop reduction of the automaton `info.rules`.

Appendix C

Contents of the CD

The attached CD contains:

- `dp_xhavle03.pdf` – text of this thesis in PDF format,
- `tex/` – the source code of this technical report in LaTeX format,
- `src/` – the source code of the implemented tool.