ADVANECD
OFRNAL LAAGGE
THOHY THOHY

Advanced Formal Language Theory

Ryan Cotterell, Anej Svete, Alexandra Butoi, Andreas Opedal, Franz Nowak

Saturday 15th July, 2023

Contents

Chapter 1

Introduction

This course serves as an in-depth introduction to the theory of weighted formal languages, a generalization of standard, unweighted, formal languages you might already be familiar with from courses such as Theory of computation. Weighted formal languages find uses in a variety of settings. In psychology, cognitive modeling, and linguistics, weighted languages can be used to model stochastic cognitive processes and behaviors (Icard, 2017, 2020). In the context of machine learning, weighted formal languages are useful abstractions for modeling distributions over structured objects, e.g., strings, trees, and graphs, and have a myriad of uses in natural language processing and computational biology. Moreover, in many cases, formal language theory provides an alternative lens through which we may view common machine learning models. For example, the well-known hidden Markov model may be viewed as either a (dynamic) graphical model or as a weighted finite-state automaton.

Advanced formal language theory is a proof-based course with a heavy implementation component, i.e., you will be expected to derive algorithms during the course and give rigorous proofs of correctness and runtime, but also implement those algorithms. During the course, you will implement most of the components found in standard libraries such as `OpenFST`[1] in the library we built for the course, `rayuela`. We will try to implement as many of the examples from the notes as possible in `Jupyter` notebooks making use of the library so that you can see examples in code directly. As you will also see, in order to maximize the similarity between the notes and the library, we have made use of Python 3's UTF-8 compatibility and placed Greek letters in the source where appropriate. Note that during coding exercises please feel free to rename these variables should typing the Greek letters slow you down. Your course assignments will include implementing parts of the library. Your solutions will be tested against our implementations.

These course notes contain all the material we will cover in the course. They are divided into chapters corresponding to the course modules and sections which roughly match the lectures. We recommend you read the corresponding sections of the notes *before* coming to the lectures and make the most of the lectures by asking questions based on your reading.

Disclaimer. This is the second time the course is being taught and we are improving the notes as we go. We will try to be as careful as possible to make them typo- and error-free. However, there will undoubtedly be mistakes scattered throughout. We will be very grateful if you report any mistakes you spot, or anything you find unclear and confusing in general—this will benefit the students as well as the teaching staff by helping us organize a better course!

[1] `https://www.openfst.org/`

Chapter 2

Preliminaries

This chapter introduces some of the basic building blocks we will be using throughout the course: semirings, weighted sets of strings, and the (weighted) Chomsky hierarchy.

2.1 Strings and Languages

First and foremost, formal language theory concerns itself with formal languages. A formal language is a set of elements that take some specified *form*. For instance, a language can be over strings, trees, or graphs. The simplest languages, and the ones we will devote the first part of the material to, are sets of *strings*.

So what is a string? We start with the notion of an alphabet.

Definition 2.1.1 (Alphabet)**.** *An alphabet, generally denoted as* Σ*, is a non-empty, finite set.*

We refer to the elements of an alphabet as **letters**.

Definition 2.1.2 (String)**.** *A* ***string***[1] *over an alphabet* Σ *is any* finite *sequence of letters from* Σ*.*

We will denote symbols in Σ with italicized lowercase letters: a, b, c, and strings using bold lowercase letters: $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$. The length of a string, written as $|\boldsymbol{y}|$, is the number of letters it contains. There is only one string of length zero, which we denote with the distinguished symbol ε and refer to as the empty string. We will assume ε is *not* an element of the original alphabet. Thus, we will sometimes write

$$\Sigma_\varepsilon \overset{\text{def}}{=} \Sigma \cup \{\varepsilon\} \tag{2.1}$$

to refer to the original alphabet Σ together with the ε symbol. As we discuss later, this makes ε one of *four* symbols with special semantics. New strings are formed from other strings and letters with concatenation.

Definition 2.1.3 (Concatenation)**.** ***Concatenation****, denoted with* $\boldsymbol{y} \circ \boldsymbol{z}$ *or just* $\boldsymbol{yz}$*, is an associative operation on strings. Formally, the concatenation of two words* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *is the word* $\boldsymbol{x} \circ \boldsymbol{y} = \boldsymbol{xy}$*, which is obtained by writing the second argument after the first one.*

The result of concatenating with ε from either side results in the original string, which means that ε is the **unit** of concatenation and the set of all words over an alphabet with the operation of concatenation forms a **monoid**.[2] This foreshadows the special role the ε symbol will play when

[1] A string is also referred to as a **word**, which continues with the linguistic terminology.

[2] We will formally introduce the notion of a unit and a monoid in §2.2.

processing strings with computational models: it will allow us to commit actions without actually reading in any symbol and moving forward in the string.

Definition 2.1.4 (Kleene Star)**.** *Let Σ be an alphabet. Define $\Sigma^0 = \{\varepsilon\}$. Now, we inductively define the set*

$$\Sigma^{n+1} = \{\boldsymbol{y}a : \boldsymbol{y} \in \Sigma^n \text{ and } a \in \Sigma\}. \tag{2.2}$$

$\boldsymbol{y}a$ denotes the concatenation *of the string $\boldsymbol{y}$ and the symbol a. The **Kleene closure** of the set Σ, denoted as Σ^*, is defined as*

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots. \tag{2.3}$$

More informally, we can think of Σ^* as the set which contains ε and all finite-length strings which can be constructed by concatenating arbitrary symbols from Σ. The Kleene closure of any alphabet is a *countably infinite* set. Importantly, notice that the Kleene closure does not contain *infinite* sequences of symbols—it is an infinite set of finite objects, just like the integers $\mathbb{Z}$.[3] The set of infinite sequences of symbols, Σ^∞, is, in contrast to Σ^*, *uncountably* infinite.[4]

Example 2.1.1 (Kleene Closure)**.** *Let $\Sigma = \{a, b, c\}$. Then*

$$\Sigma^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, aac, \dots\}. \tag{2.4}$$

Definition 2.1.5 (String Language)**.** *A **string language** L over an alphabet Σ is a subset of Σ^*. Thus, L is a set of sequences of elements of Σ, which are generally termed **words**.*

Finally, we introduce two notions of subelements of strings.

Definition 2.1.6 (Subelements of strings)**.** *A **subsequence** of a string $\boldsymbol{y}$ is defined as a sequence that can be formed from $\boldsymbol{y}$ by deleting some or no letters, leaving the order untouched. A **substring** is a contiguous subsequence.**Prefixes** and **suffixes** are special cases of substrings. A prefix is a substring of $\boldsymbol{y}$ that shares the same first letter as $\boldsymbol{y}$ and a suffix is a substring of $\boldsymbol{y}$ that shares the same last letter as $\boldsymbol{y}$.*

For instance, ab and bc are substrings and subsequences of $\boldsymbol{y} = abc$, while ac is a subsequence but not a substring.

[3] Indeed, in the scope of the course (and in computer science in general), a string is always a *finite* sequence of symbols.

[4] Note that we will *not* deal with infinite sequences later in the notes.

2.2 Semirings

One of the central themes of the course will be the *weighting* of formal languages. The entirety of the course will discuss computational models to represent and manipulate weighted languages. Weighting, for example, allows us to assign weights to strings in a language, analogously to how we can endow a formal language with probabilities. We define weighted (string) languages in §2.3. However, in order to formally discuss weights in the appropriate generality, we first need a model of what weights *are* and what we can *do* with them. As it turns out, there is a useful algebraic structure for describing weights that admit tractable computation: semirings. Semirings are an abstract algebraic concept (a *universal algebra*, cf. **??**) which provides a very rigorous yet flexible mathematical framework for defining such weightings. This subsection introduces them and gives a few examples of how we will be using them in this course.

Before we talk about semirings, however, we have to familiarize ourselves with two more basic algebraic structures: monoids and groups.

Definition 2.2.1. *A **monoid** is a set equipped with an associative binary operation and an identity element; let $\mathbb{K}$ be a set and $\odot$ a binary operation on it. Then the tuple $(\mathbb{K}, \odot)$ is a monoid with the identity element $\mathbf{1} \in \mathbb{K}$ if the following properties hold:*

(i) *$\odot$ is closed in $\mathbb{K}$: $\forall a, b \in \mathbb{K} : a \odot b \in \mathbb{K}$;*

(ii) *$\odot$ is associative: $\forall a, b, c \in \mathbb{K} : (a \odot b) \odot c = a \odot (b \odot c)$;*

(iii) *$\mathbf{1}$ is the left and right unit: $\forall a \in \mathbb{K} : \mathbf{1} \odot a = a \odot \mathbf{1} = a$;*

(iv) *$\mathbb{K}$ is closed under $\odot$: $\forall a, b \in \mathbb{K} : a \odot b \in \mathbb{K}$.*

*A monoid is **commutative** if additionally $\forall a, b \in \mathbb{K} :\ a \odot b = b \odot a$. A **group** is a monoid with one additional property: every element of $\mathbb{K}$ has an inverse, i.e, $\forall a \in \mathbb{K}\ \exists b \in \mathbb{K} : b \odot a = a \odot b = \mathbf{1}$. It is tradition to denote this b as a^{-1}.*

Although the terms monoid and group strictly refer to the *tuple* $(\mathbb{K}, \odot)$, the set $\mathbb{K}$ is often referred as the monoid/group on its own, if the operation is clear from the context.

These two building blocks allow us to define a semiring.

Definition 2.2.2. *A **semiring** $\mathcal{W}$ is a 5-tuple $(\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, where $\mathbb{K}$ is a set equipped with two binary operations $\oplus$ and $\otimes$, such that:*

(i) *$(\mathbb{K}, \oplus)$ is a **commutative monoid** with identity element $\mathbf{0}$, i.e., $\forall a, b, c \in \mathbb{K}$:*

- $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- $\mathbf{0} \oplus a = a \oplus \mathbf{0} = a$
- $a \oplus b = b \oplus a$

(ii) *$(\mathbb{K}, \otimes)$ is a **monoid** with identity element $\mathbf{1}$, i.e., $\forall a, b, c \in \mathbb{K}$:*

- $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
- $\mathbf{1} \otimes a = a \otimes \mathbf{1} = a$

(iii) *Multiplication left and right **distributes** over addition:*

- $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$

- $(b \oplus c) \otimes a = b \otimes a \oplus c \otimes a$

(iv) Multiplication with ***0 annihilates*** $\mathbb{K}$*, i.e.,* $\forall a \in \mathbb{K}$*:*

- $\boldsymbol{0} \otimes a = a \otimes \boldsymbol{0} = \boldsymbol{0}$

This is a very abstract definition. The notation for operations $\oplus$ and $\otimes$ is completely general and actual examples of semirings will often "overwrite" the symbols with specific ones (see below). The symbol for multiplication is often left out, i.e., $a \otimes b = ab$.

A semiring is a weaker notion than the similar **ring**, which differs from a semiring by requiring the addition to be *invertible*, which means that the structure $(\mathbb{K}, \oplus)$ has to be a commutative *group* rather than a commutative monoid. There are also a few special classes of semirings that impose different restrictions on the operations and thus allow us to operate on them more specifically. If the multiplicative operation is also *commutative*, the semiring is called a **commutative semiring**. If addition is idempotent, i.e., $\forall a \in \mathbb{K}, a \oplus a = a$, the semiring is **idempotent**. We will encounter more special semirings throughout the course.

We now introduce some commonly used semirings and list some applications where they appear.

Example 2.2.1 (Boolean semiring)**.** *The* ***boolean semiring*** *is a semiring where* $\mathbb{K} = \{0, 1\}$*,* $\oplus = \vee$*,* $\otimes = \wedge$*,* $\boldsymbol{0} = 0$*, and* $\boldsymbol{1} = 1$*. It is the most basic semiring and is used to test for acceptance of strings by automata.*

Example 2.2.2 (Real semiring)**.** *The* ***real semiring*** *is a semiring where* $\mathbb{K} = \mathbb{R}$*,* $\oplus = +$*,* $\otimes = \cdot$*,* $\boldsymbol{0} = 0$*, and* $\boldsymbol{1} = 1$*. It is probably the semiring you are most familar with, as it indeed represents the reals with the standard addition and multiplication. The restriction of the set to* $[0, 1]$ *gives us the* ***probability semiring*** *which we will come across often.*

Example 2.2.3 (Tropical semiring)**.** *The* ***tropical semiring*** *is a semiring where* $\mathbb{K} = \mathbb{R}_+ \cup \{\infty\}$*,* $\oplus = \min$*,* $\otimes = +$*,* $\boldsymbol{0} = \infty$*, and* $\boldsymbol{1} = 0$*. Its "opposite" is the* ***max-plus****, or simply the* ***arctic****, semiring, where* $\mathbb{K} = \mathbb{R}_- \cup \{-\infty\}$*,* $\oplus = \max$*,* $\otimes = +$*,* $\boldsymbol{0} = -\infty$*, and* $\boldsymbol{1} = 0$*. Note that* ∞ *and* $-\infty$ are not *elements of* $\mathbb{R}$ *but rather special entities with specific semantics. These two semirings are often used when computing the minimum/maximum values over some set of values, for example, the accepting paths of a string (this will be formally defined shortly).*

The boolean, real, and tropical semirings are commutative. The boolean and tropical ones are also idempotent. A few more useful semirings are listed in Tab. 2.1. You can find a number of implementations of semirings in rayuela. A few examples of how to use them and work with them are also presented in the accompanying Jupyter notebook.

Semiring	$\mathbb{K}$	$\oplus$	$\otimes$	**0**	**1**
Probability	$[0, 1]$	$+$	$\cdot$	0	1
String	Σ^*	longest common prefix	concatenation	"∞"	ε
Log	$\mathbb{R} \cup \{-\infty, \infty\}$	$\oplus_{\log}$	$+$	∞	0
Rational	$\mathbb{Q}$	$+$	$\cdot$	0	1

Table 2.1: Examples of some other common semirings. $\oplus_{\log}$ is defined as $x \oplus_{\log} y = -\log\left(e^{-x} + e^{-y}\right)$. The "$\infty$" is a special entity, which is importantly *not* in Σ^*—the same way that $\pm\infty \notin \mathbb{R}$. It is defined such that the longest common prefix (lcp) of any string $\boldsymbol{y}$ with "∞" is $\boldsymbol{y}$. Here we assume multiplication only left distributes over addition.

2.3 Weighted Sets of Strings as Formal Power Series

A weighted formal language L over a semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a generalization of a formal language where, instead of indicating whether a string $\boldsymbol{y}$ is in L, we assign every string $\boldsymbol{y} \in \Sigma^*$ a *weight*—an element from $\mathbb{K}$. This naturally generalizes the notion of an unweighted language—we recover the unweighted languages through the boolean semiring. As we will see, weighted formal languages arise naturally from weighted computational models in formal language theory, which we start introducing from Ch. 3 on.

A Gentle Introduction to Representing Weighted Formal Languages as Formal Power Series

As we discussed, a formal language is a set of strings—a subset fo Σ^* for some alphabet Σ. Intuitively, you can thing of a *weighted* formal language as a (possibly infinite) set of string-weight *pairs* $(\boldsymbol{y}, w(\boldsymbol{y}))$.

Example 2.3.1. *The set $L = \{y^n \mid n \in \mathbb{N}_{\geq 0}\}$ is an instance of an unweighed formal language. We can now define a function $f(y^n) = 2^{-n}$ and with it specify the* weighted *language $L' = \{(y^n, f(y^n)) \mid n \in \mathbb{N}_{\geq 0}\} = \{(y^n, 2^{-n}) \mid n \in \mathbb{N}_{\geq 0}\}$. Here, $y^n = \underbrace{y \cdots y}_{n\text{-times}}$.*

A more formal, yet still very natural and mathematically convenient object for representing weighted formal languages, is *formal power series*, which we introduce next. Formal power series are a generalization of polynomials to objects with an infinite number of terms. A formal power series with the variable x might look something like

$$r(x) = \sum_{n=0}^{\infty} a_n x^n, \tag{2.5}$$

where a_n are its **coefficients** and x^n the **terms** of the series. Formal power series are close cousins of infinite power series like $\sum_{n=0}^{\infty} q^n$ that one learns in basic calculus, but come with a crucial caveat—in contrast to normal power series, where we are concerned with the notion of convergence and *evaluating* the value of the power series at specific input values, we consider formal series algebraically *without* the need to evaluate them. For example, the value of the power series $\sum_{n=0}^{\infty} q^n$ is $\frac{1}{1-q}$ for $q \in (0, 1)$. On the other hand, the formal power series from Eq. (2.5) has *no* value—we never "insert" any specific value for x. The series simply exists as an object with can be manipulated and examined. This might be hard to grasp if you have not seen formal power series before. In that case, it might be helpful to think of the individual terms x^n in the formal power series as "*hangers*" which keep the coefficients a_n at the appropriate place in the series such that they can be manipulated and "accessed" conveniently with algebraic operations.

So, why are we interested in formal power series? It turns out they are a very handy formalism (you could even simply say notation) for formally specifying weighted formal languages—we will simply specify a weighted formal language with a formal power series. In this case, the role of the terms x^n will be taken by taken by *strings* in Σ^* (with some enumeration of Σ^*), so they will, informally, looks something like

$$r = \sum_{\boldsymbol{y} \in \Sigma^*} a_n \boldsymbol{y}, \tag{2.6}$$

where the sum will "order" the strings in some way (this can always be done, since Σ^* is a countably infinite set). Again, since we are not interested in evaluating the series at any point, the role of the terms that the coefficients are associated with is simply to be there, “hooked” to the coefficients—therefore, they may as well be strings. In the case of weighted formal languages, the coefficients hooked to the terms—strings—will be the *weights* of those strings in the formal language. For example, the weighted language over the real semiring from Example 2.3.1 can be expressed as the following formal power series

$$\sum_{\boldsymbol{y}\in\{y\}^*} f\left(\boldsymbol{y}\right)\boldsymbol{y} = \sum_{n=1}^{\infty} 2^{-n} y^n. \tag{2.7}$$

This is very similar to simply specifying the set $\{(y^n, 2^{-n}) \mid n \in \mathbb{N}_{\geq 0}\}$ we specified above. The only difference is that the strings are now ordered and fixed in their place in the formal power series.

Here is why describing weighted formal languages with formal power series is useful. Just like classical power series, formal power series can be manipulated with the usual algebraic operations on series (addition, subtraction, multiplication, division, partial sums, etc.). These will then naturally and conveniently correspond to operations on the underlying *languages* that they represent. Indeed, we employ the summand notation in formal power series as it is suggestive of the fact that these properties hold. To reiterate, it is crucial to keep in mind we are *not* interested in convergence or the value of a formal power series as we would be if we were studying power series over the reals. The coefficients of individual terms in a formal power series are used simply as placeholders for weights associated with elements of a potentially infinite set.

Formal Definition of Formal Power Series

Let us next dive into how they will be useful in the context of the notes more formally. We start by defining the weighted language semiring.

Definition 2.3.1 (Weighted Language Semiring)**.** *Let Σ be an alphabet and $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ a semiring of weights. The* ***weighted language semiring*** *$\mathcal{W}^{\Sigma^*}$ is the semiring*

$$\mathcal{W}^{\Sigma^*} = \left(\mathbb{K}^{\Sigma^*}, \oplus_{\mathrm{L}}, \otimes_{\mathrm{L}}, \boldsymbol{0}_L, \boldsymbol{1}_L\right), \tag{2.8}$$

where $\mathbb{K}^{\Sigma^*} = \Sigma^* \to \mathbb{K}$ is the ***set of all functions*** *from Σ^* to $\mathbb{K}$. $\oplus_{\mathrm{L}}$ (the addition of two functions) is defined as*

$$(v \oplus_{\mathrm{L}} w)(\boldsymbol{y}) \overset{\text{def}}{=} v(\boldsymbol{y}) \oplus w(\boldsymbol{y}) \tag{2.9}$$

for $v, w \in \mathbb{K}^{\Sigma^*}, \boldsymbol{y} \in \Sigma^*$, and $\otimes_{\mathrm{L}}$ (the multiplication of two functions) is defined as

$$(v \otimes_{\mathrm{L}} w)(\boldsymbol{y}) \overset{\text{def}}{=} \bigoplus_{\boldsymbol{y}_1\boldsymbol{y}_2=\boldsymbol{y}} v(\boldsymbol{y}_1) \otimes w(\boldsymbol{y}_2) \tag{2.10}$$

for $v, w \in \mathbb{K}^{\Sigma^*}, \boldsymbol{y}, \boldsymbol{y}_1, \boldsymbol{y}_2 \in \Sigma^*$. Lastly, the units of the semiring (which are function) are defined as

$$\boldsymbol{0}_L : \boldsymbol{y} \mapsto \boldsymbol{0} \tag{2.11}$$

$$\boldsymbol{1}_L : \boldsymbol{y} \mapsto \begin{cases} \boldsymbol{1}, & \boldsymbol{y} = \varepsilon \\ \boldsymbol{0}, & \textit{otherwise} \end{cases} \tag{2.12}$$

To reiterate, the elements of $\mathbb{K}^{\Sigma^*}$ are *function*—any element of $\mathbb{K}^{\Sigma^*}$, which we will generally denote by r, assigns any string in Σ^* a weight in $\mathbb{K}$, just like $f\left(y^n\right) = 2^{-n}$ assigned a weight to any string in $\{y\}^*$. This allows us to formally define a formal power series for weighted languages.

Definition 2.3.2 (Formal Power Series)**.** *Let Σ be an alphabet $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ a semiring, and $\mathcal{W}^{\Sigma^*}$ the weighted language semiring with weights from $\mathcal{W}$. Elements r of $\mathcal{W}^{\Sigma^*}$, i.e., mappings from Σ^* to $\mathbb{K}$, are called **formal power series**. The* values *of r are denoted by $r(\boldsymbol{y})$ where $\boldsymbol{y} \in \Sigma^*$. Instead of specifying the function (formal power series) r as a mapping $\boldsymbol{y} \mapsto r(\boldsymbol{y})$, r is written as a summation:*

$$r \overset{\text{def}}{=} \sum_{\boldsymbol{y} \in \Sigma^*} r(\boldsymbol{y})\boldsymbol{y} \overset{\text{def}}{=} \bigcup_{\boldsymbol{y} \in \Sigma^*} \{r(\boldsymbol{y})\boldsymbol{y}\}, \tag{2.13}$$

*where we assume some fixed ordering of Σ^*. The values $r(\boldsymbol{y})$ are called the **coefficients** of the series—$r(\boldsymbol{y})$ is the "weight" assigned by r to the element $\boldsymbol{y}$.*

Notice the two different "roles" of r—as a function $\Sigma^* \to \mathbb{K}$, it is denoted by the summation $r = \sum_{\boldsymbol{y} \in \Sigma^*} r(\boldsymbol{y})\boldsymbol{y}$. On the other hand, it also defines the coefficients in the sum—those are the result of applying the function to a specific string, i.e., $r(\boldsymbol{y})$.

Note that the sum $\sum$ is little more than notation in this treatment. Instead of the powers of the indeterminate as in a normal power series (e.g., x), the coefficients in a formal power series weight all strings $\boldsymbol{y} \in \Sigma^*$. The collection of all formal power series r, as defined above, is denoted by $\mathcal{W}\langle\langle\Sigma^*\rangle\rangle$. Again, we emphasize the intuitive understanding you should have about formal power series. We are not interested in *summing up* the series but rather in their use in representing languages. Indeed, the primary function of formal power series in the study of weighted automata lies in their ability to serve as a useful *notation* for representing weighted sets of strings.

Example 2.3.2 (A Geometric Series)**.** *As a contrast to* formal *power series, consider the* geometric *series $1 + \frac{1}{2}x + \frac{1}{4}x^2 + \frac{1}{8}x^3 + \frac{1}{16}x^4 + \cdots = \sum\limits_{n=0}^{\infty} \left(\frac{1}{2}\right)^n x^n$. We see that it converges for $|x| < 2$, so we can actually calculate its value for any $x \in (-2, 2)$.*

Alternatively, we could also interpret it as a formal power series in which the different x^n represent elements of some infinite set and not an indeterminate. In that case, we would not evaluate the series at any value (since x is not *an indeterminate!) but rather understand it as the notation for saying that the element with the identified x^n receives the weight $\frac{1}{2^n}$ by the formal power series.*

Definition 2.3.3 (Support of a Formal Power Series)**.** *Given a $r \in \mathcal{W}\langle\langle\Sigma^*\rangle\rangle$, the **support** of r is the set*

$$\operatorname{supp}(r) = \{\boldsymbol{y} \in \Sigma^* \mid r(\boldsymbol{y}) \neq \boldsymbol{0}\}. \tag{2.14}$$

Definition 2.3.4 (Characteristic Series)**.** *A series $r \in \mathcal{W}\langle\langle\Sigma^*\rangle\rangle$ is the **characteristic series** of its support L if every coefficient of r equals $\boldsymbol{0}$ or $\boldsymbol{1}$. We denote that as $r = \operatorname{char}(L) = \mathbb{1}\{L\}$.*

This makes the connection to weighted sets of strings clear—the (infinite) sum represents the set of strings $\boldsymbol{y}$ *together* with their weights $r(\boldsymbol{y})$ directly! Characteristic series correspond to *unweighted* finite-state automata as a way to specify whether a string is an element of a language or not. The subset $\mathcal{W}\langle\Sigma^*\rangle \subseteq \mathcal{W}\langle\langle\Sigma^*\rangle\rangle$ denotes all series with *finite support*, which are, going with the intuition, called **polynomials**.

Example 2.3.3. *Examples of polynomials in $\mathcal{W}\langle\Sigma^*\rangle$ are $\boldsymbol{0}$ and $x\boldsymbol{y}$ where $x \in \mathbb{K}$ and $\boldsymbol{y} \in \Sigma^*$. The $\boldsymbol{0}$ polynomial is defined as $\boldsymbol{0}(\boldsymbol{y}) = \boldsymbol{0}\ \forall \boldsymbol{y}$, whereas the $x\boldsymbol{y}$ polynomial is defined as $(x\boldsymbol{y})(\boldsymbol{y}) = x$ and $(x\boldsymbol{y})(\boldsymbol{y}') = \boldsymbol{0}\ \forall \boldsymbol{y}' \neq \boldsymbol{y}$. The support of the polynomial $x\boldsymbol{y}$ is thus the singleton $\{\boldsymbol{y}\}$, which is weighted with x. $\boldsymbol{1y}$ is often denoted just by $\boldsymbol{y}$ or $\mathbb{1}\{\boldsymbol{y}\}$, which makes sense, as it acts like an indicator function, taking the value $\boldsymbol{1}$ if and only if the argument is $\boldsymbol{y}$.*

For example, the coefficients of the formal power series $r = 3.14cab \in \mathcal{W}\langle\langle\Sigma^*\rangle\rangle$ for $\Sigma = \{a, b, c\}$ over the real semiring are $r(cab) = 3.14$ and $r(\boldsymbol{y}) = 0$ for any other $\boldsymbol{y} \neq cab$.

The formal definition of the set in this way also means that we can define a number of useful operations on it. We conclude by listing some common operations under which $\mathcal{W}\langle\langle\Sigma^*\rangle\rangle$ is closed.

(i) **addition**: $\sum_{\boldsymbol{y}\in\Sigma^*} v(\boldsymbol{y})\boldsymbol{y} + \sum_{\boldsymbol{y}\in\Sigma^*} w(\boldsymbol{y})\boldsymbol{y} = \sum_{\boldsymbol{y}\in\Sigma^*} (v(\boldsymbol{y}) \oplus w(\boldsymbol{y}))\boldsymbol{y}$

(ii) **Cauchy product**: $\sum_{\boldsymbol{y}\in\Sigma^*} v(\boldsymbol{y})\boldsymbol{y} * \sum_{\boldsymbol{y}\in\Sigma^*} w(\boldsymbol{y})\boldsymbol{y} = \sum_{\boldsymbol{y}\in\Sigma^*} \bigoplus_{\boldsymbol{y}_1\boldsymbol{y}_2=\boldsymbol{y}} (v(\boldsymbol{y}_1) \otimes w(\boldsymbol{y}_2))\boldsymbol{y}$

(iii) **pointwise multiplication (Hadamard product)**: $\sum_{\boldsymbol{y}\in\Sigma^*} v(\boldsymbol{y})\boldsymbol{y} \odot \sum_{\boldsymbol{y}\in\Sigma^*} w(\boldsymbol{y})\boldsymbol{y} = \sum_{\boldsymbol{y}\in\Sigma^*} (v(\boldsymbol{y}) \otimes w(\boldsymbol{y}))\boldsymbol{y}$

(iv) **left scalar product**: $x \times \left(\sum_{\boldsymbol{y}\in\Sigma^*} w(\boldsymbol{y})\boldsymbol{y}\right) = \sum_{\boldsymbol{y}\in\Sigma^*} (x \otimes w(\boldsymbol{y}))\boldsymbol{y}$

(v) **right scalar product**: $\left(\sum_{\boldsymbol{y}\in\Sigma^*} w(\boldsymbol{y})\boldsymbol{y}\right) \times x = \sum_{\boldsymbol{y}\in\Sigma^*} (w(\boldsymbol{y}) \otimes x)\boldsymbol{y}$

The Cauchy product is the discrete **convolution** of two infinite series, which inspired our naming of the operator $*$. Scalar multiplication is also just a (common) special case of the Cauchy product, namely with the formal power series $x\varepsilon$ (from matching sides). These operations have natural analogs in terms of operations on finite-state machines and other devices, which we start diving into shortly. We will see that

- power series addition corresponds to weighted union;
- the Cauchy product corresponds to weighted concatenation;
- pointwise multiplication corresponds to weighted intersection;
- left scalar multiplication corresponds to left-multiplying the initial weights by a scalar in a weighted finite-state automaton;
- right scalar multiplication corresponds to right-multiplying the final weights by a scalar in a weighted finite-state automaton.

Example 2.3.4. *To see a few operations in action, let us consider the formal power series* $r = 2 \times (3ba \oplus 4ba \oplus (b \times 8)) \in \mathcal{R}\langle\langle\{b,a\}^*\rangle\rangle$ *over the real semiring and try to simplify it.*

$$\begin{aligned}
r &= 2 \times (3ba + 4ba + (b \times 8)) && \\
&= 2 \times (3ba + 4ba + 8b) && \textit{(scalar product)} \\
&= 2 \times (7ba + 8b) && \textit{(addition)} \\
&= 2 \times 7ba + 2 \times 8b && \textit{(distributivity)} \\
&= 14ba + 16b && \textit{(scalar product)}
\end{aligned}$$

The resulting power series thus represents the weighted formal language that assigns the string ba *the weight* 14 *and the string* b *the weight* 16.

2.4 Languages as Systems of Equations

Throughout this course, we will see several ways of characterizing languages. For instance, regular languages are sets of strings that are recognized by finite-state automata or specified using a regular expression. Context-free languages, on the other hand, are languages that can be generated by context-free grammars or recognized by pushdown automata. Before we dive into the details of these formalisms, we'll look at another way of characterizing languages, namely *systems of equations.* As we will see next, regular languages are languages that are solutions to systems of *linear* equations while context-free languages are solutions to systems of *polynomial* equations.

Definition 2.4.1. *Let Σ be an alphabet and $\mathcal{W}$ a semiring of weights. Additionally, $\mathcal{W}^{\Sigma^*}$ is the weighted language semiring over the alphabet Σ with weights from $\mathcal{W}$. A system of linear equations over the weighted language semiring is a collection of equations of the form*

$$X = a_1 \otimes_L Y_1 \oplus_L a_2 \otimes_L Y_2 \oplus_L \dots a_n \otimes_L Y_n, \tag{2.15}$$

where $a_1, \dots, a_n$ are elements of $\mathcal{W}^{\Sigma^*}$ and $X, Y_1, \dots, Y_n$ are variables.

To simplify the notation, we will use $\oplus$ and $\otimes$ in the remainder of this section to denote addition and multiplication in the weighted language semiring.

Example 2.4.1. *Consider the simple alphabet $\Sigma = \{a\}$ and the boolean semiring $\mathcal{W}$. The system of linear equations*

$$\begin{aligned} Y &= \varepsilon \\ X &= (a \otimes Y) \oplus (a \otimes X) \end{aligned}$$

has as unique solution the regular language $X = \{a, aa, aaa, \dots\}$.

We give the following theorem without proof for now, since it requires some machinery that we haven't encountered yet. We will come back to it in a later section, once we've presented finite-state automata.

Theorem 2.4.1. *A language is regular iff it is the solution to a system of linear equations over the weighted language semiring.*

It turns out that linear equations are expressive enough to characterize regular languages only. Can we still use systems of equations to characterize other types of languages? Yes! Context-free languages can be thought of as solutions to systems of polynomial equations, like the one from the following example.

Example 2.4.2. *Consider the alphabet $\Sigma = \{a, b\}$ and the boolean semiring $\mathcal{W}$. The system of equations*

$$\begin{aligned} X &= a \otimes (X \otimes Y) \oplus \varepsilon \\ Y &= b \otimes Z \\ Z &= \varepsilon \end{aligned}$$

has as solution the context-free language $X = \{a^n b^n \mid n \geq 0\}$.

Chapter 3

Regular Languages

3.1 Finite-state automata

We will now define the first computational tool that will allow us to describe basic sets of strings and answer certain questions about them. This tool is the finite-state automaton. We will start with a formal definition and then look at some examples. A **finite-state automaton**[1] is a computational device that determines whether a string is an element of a given language. Another way to think of it would be as a system to specify a set of rules strings must satisfy to be included in the language.

Definition 3.1.1. *A* ***finite-state automaton*** *is a 5-tuple* $(\Sigma, Q, I, F, \delta)$ *where*

- Σ *is an alphabet;*
- Q *is a finite set of states;*
- $I \subseteq Q$ *is the set of initial states;*
- $F \subseteq Q$ *the set of final or accepting states;*
- *A finite multi-set* $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$, [2] *where elements of* δ *are generally called* ***transitions****.*

The name, *finite-state* automaton, stems from the requirement that the set of states is finite; we will later in the course show how context-free grammars, by conversion to their automatic analogs, can be viewed as infinite state machines.

We will denote a general automaton with a (subscripted) $\mathcal{A}$. In the next sections, we will refer to the **size** of a general automaton $\mathcal{A}$ as the sum of the number of states and transitions in the automaton, i.e. $|\mathcal{A}| \overset{\text{def}}{=} |Q| + |\delta|$.

We define an FSA with a set of initial states for generality. Often, they are defined with a single initial state, which is equivalent, since a machine with multiple initial states can be represented as one with only one.

The transition "function" δ can return more than one state to transition to in the non-deterministic case, which will be formally described shortly. An alternative definition you can come across defines δ as $\delta : Q \times \Sigma \to Q$ in the deterministic case and $\delta : Q \times \Sigma \to 2^Q$ in the non-deterministic case. To avoid the need to distinguish the two cases and to make the definition more similar to the one of *weighted* FSA later, we will use the multi-set-based definition.

[1] The plural of automaton is automata—it is a second-declension Greek noun after all.

[2] The fact that it is a multi-set reflects that it can contain multiple copies of the same element (i.e., transitions between the same pair of states with the same symbol).

An FSA can be graphically represented as a labeled, directed multi-graph.[3] The vertices in the graph represent the states $q \in Q$ and the (labeled) edges between them the transitions in δ. The labels on the edges correspond to the input symbols $a \in \Sigma$ which are consumed when transitioning over the edges. The initial states $q_I \in I$ are marked by a special incoming arrow while the final states $q_F \in F$ are indicated using a double circle instead of a single one.

Example 3.1.1. *An example of an FSA can be seen in Fig. 3.1. Formally, we can specify it as*

- $\Sigma = \{a, b, c\}$
- $Q = \{1, 2, 3\}$
- $I = \{1\}$
- $F = \{3\}$
- $\delta = \{(1, a, 2), (1, b, 3), (2, b, 2), (2, c, 3)\}$

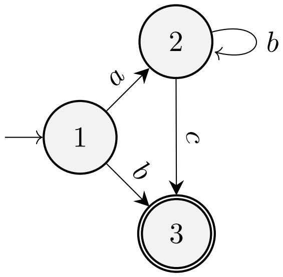


Figure 3.1: Example of a simple FSA.

The automaton sequentially reads in individual symbols of an **input string** $\boldsymbol{y} \in \Sigma^*$ and transitions from state to state according to the transition function δ. We start a run through the automaton in a starting state $q_I \in I$ (more precisely, we act as if we start in all of them in parallel). The automaton transitions from state q into the state q' upon reading the symbol a if and only if $(q, a, q') \in \delta$. If the automaton ends up, after reading in the last symbol of the input string, in one of the final states $q_F \in F$, we say that the automaton *recognizes* that string.[4]

A natural question to ask at this point is what happens if for a state–symbol pair (q, a) there is *more than one* possible transition allowed under the relation δ. In such a case, we take *all* implicit transitions simultaneously. This leads us to a pair of definitions.

Definition 3.1.2. *A FSA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ is* ***deterministic*** *or* ***sequential*** *if*

- *it does not have any ε-transitions;*
- *for every $(q, a) \in Q \times \Sigma$, there is at most one $q' \in Q$ such that $(q, a, q') \in \delta$;*
- *there is a single initial state i.e., $|I| = 1$.*

Otherwise, we say $\mathcal{A}$ is ***non-deterministic****.*

[3] The *multi-* aspect of the multi-graph refers to the fact that we can have multiple arcs from any pair of states and labeled refers to the fact that we label those arcs with symbols from the alphabet Σ.

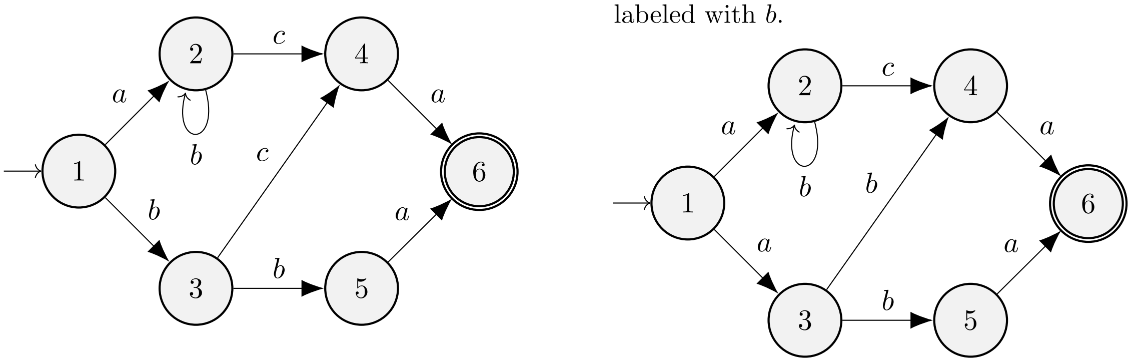
[4] We will make the notion of recognition more formal soon.

An important, and perhaps not entirely obvious, result we will show soon is that the classes of deterministic and non-deterministic FSA are *equivalent*, in the sense that you can always represent a member of one class with a member of the other. Throughout the text, we will also adopt a more suggestive notation for transitions by denoting a transition (q_1, a, q_2) as $q_1 \xrightarrow{a} q_2$.

Example 3.1.2. *Some more simple examples of FSAs are shown in Fig. 3.2. The FSA in Fig. 3.2a, for example, can formally be defined with*

- $\Sigma = \{a, b, c\}$
- $Q = \{1, 2, 3, 4, 5, 6\}$
- $I = \{1\}$
- $F = \{6\}$
- $\delta = \{(1, a, 2), (1, b, 3), (2, b, 2), (2, c, 4), (3, c, 4), (3, b, 5), (4, a, 6), (5, a, 6)\}$

The FSA in Fig. 3.2a is deterministic while the one in Fig. 3.2b is non-deterministic.



(a) A deterministic FSA, $\mathcal{A}_1$. Each state only has one outgoing transition labeled with the same symbol.

(b) A non-deterministic FSA, $\mathcal{A}_2$. State 1 has two outgoing transitions labeled with a whereas state 3 has two outgoing transitions labeled with b.

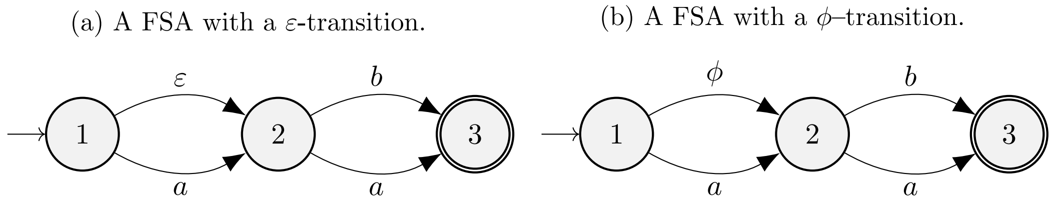
Figure 3.2: Examples of a deterministic and a non-deterministic FSA.

Automaton vs. Acceptor vs. Machine vs. Transducer Throughout the text, we will refer to (weighted) finite-state automata interchangeably as (weighted) finite-state *machines* and (weighted) finite-state *acceptors*. Note that the latter is more specific—it refers to automata that explicitly operate on a single tape (i.e., on a single input string). Soon, we will extend that to two-tape-based finite-state machines, called (weighted) finite-state *transducers*. Keep in mind, however, that we usually use the general term automaton where the form of the input (number of tapes) is apparent.

3.1.1 Special Symbols

In this course, we are going to cover four symbols with special semantics. They are ε, ρ (for *rest*), ϕ (for *failure*) and σ. We will introduce each of the symbols with an example in turn. The first symbol, ε, allows a finite-state machine to transition to a new state *without* consuming a symbol. This is in line with its definition as an empty string. The second symbol, ϕ, allows the machine to transition to a new state *without* consuming a symbol, *but only when no other valid transition is possible.*

Example 3.1.3. *Below is an example that contrasts ε and ϕ. Consider the pair of two-state FSA on the alphabet $\Sigma = \{a, b\}$, shown in Fig. 3.3. The automaton in Fig. 3.3a accepts the language $\{a, b, aa, ab\}$, while the automaton in Fig. 3.3b accepts the language $\{b, aa, ab\}$. The latter will not accept the string a since it is forced to take the transition $1 \xrightarrow{a} 2$ upon reading a, and is then stuck in the state 2. Upon reading b, on the other hand, it is allowed to jump to 2 without consuming the input symbol and thus can end up in the final state 3.*


(a) A FSA with a ε-transition.

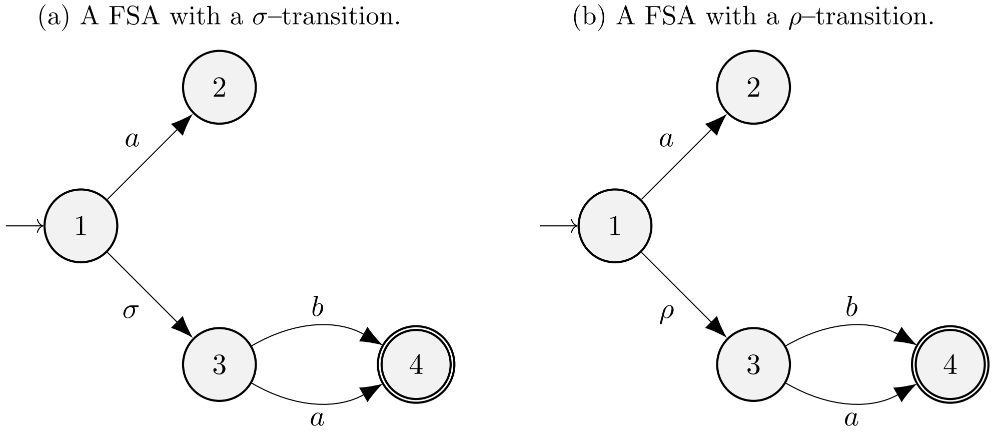
(b) A FSA with a ϕ–transition.

Figure 3.3: FSA with ε and ϕ transitions.

At first blush, it might not be obvious why we would want to employ a ϕ transition. Indeed, one may simulate a ϕ transition, by adding additional transitions to the machine. The motivation for the development of ϕ-transitions lies in computational efficiency: ϕ-transitions allow for a more compact representation of certain languages (Allauzen et al., 2003).

Next, we discuss the special symbol σ, which may always be taken, as ε, but in contrast to ε, *consumes* the next symbol in the string. We contrast σ with ρ, which consumes a symbol, but *may only be taken if no other option is available.* Again, it is easiest to see how σ and ρ function through an example.

Example 3.1.4. *We now also give an example that contrasts σ and ρ. Consider the pair of four-state FSA on the alphabet $\Sigma = \{a, b\}$, shown in Fig. 3.4. The automaton in Fig. 3.4a accepts the language $\{aa, ab, ba, bb\}$. Since σ consumes a symbol in any case, any accepted word has to be of length at least 2. Finally, the automaton in Fig. 3.4b accepts the language $\{ba, bb\}$, since, again, the automaton is forced to consume and move to the "dead–end" state 2 whenever the input string starts with a.*


(a) A FSA with a σ–transition.

(b) A FSA with a ρ–transition.

Figure 3.4: FSA with σ and ρ transitions.

The most commonly employed symbol is ε. The symbols ϕ, ρ and σ were introduced in the 2000s to more easily encode various NLP models as finite-state machines (Allauzen et al., 2003).

Similarly to how we included ε in the definition of δ in Def. 3.1.2, we could include the other 3 special characters as well.

3.1.2 Paths

The concept of walks in the transition graph of an FSA, so far addressed in an intuitive and informal manner, can be treated more rigorously by formally defining **paths**.

Definition 3.1.3 (Path)**.** *A* ***path*** $\boldsymbol{\pi}$ *is an element of* δ^* *with* consecutive transitions*, meaning that it is of the form* $\left(q_1 \xrightarrow{\bullet} q_2, q_2 \xrightarrow{\bullet} q_3 \dots q_{n-1} \xrightarrow{\bullet} q_n\right)$*, where* $\bullet$ *is a placeholder.*[5] *The* ***length*** *of a path is the number of transition in it; we denote the length as* $|\boldsymbol{\pi}|$*. The* ***yield*** *of a path is the concatenation of the input symbols on the edges along the path, which we will mark with* $\mathrm{s}\left(\boldsymbol{\pi}\right)$*. Furthermore, we denote sets of paths with capital* Π*.*

Throughout the text, we will use a few different variants involving Π *to avoid clutter:*

- $\Pi(\mathcal{A})$ *as the set of all paths in automaton* $\mathcal{A}$*;*
- $\Pi(\boldsymbol{y})$ *as the set of all paths with yield* $\boldsymbol{y} \in \Sigma^*$*;*
- $\Pi(q)$ *as the set of all paths starting at state* $q \in Q$*;*
- $\Pi(q, q')$ *as the set of all paths from state* q *to* q'*;*
- $\Pi(q, \boldsymbol{y}, q')$ *the set of all paths from state* q *to* q' *with the yield* $\boldsymbol{y}$
- $\Pi(\mathcal{Q}) = \bigcup\limits_{q \in \mathcal{Q}} \Pi\left(q\right)$*;*
- $\Pi(\mathcal{Q}, \mathcal{Q}') = \bigcup\limits_{q \in \mathcal{Q}, q' \in \mathcal{Q}'} \Pi\left(q, q'\right)$*;*
- $\Pi(\mathcal{Q}, \boldsymbol{y}, \mathcal{Q}') = \bigcup\limits_{q \in \mathcal{Q}, q' \in \mathcal{Q}'} \Pi\left(q, \boldsymbol{y}, q'\right)$*;*

Definition 3.1.4 (Cyclic FSA)**.** *We say a path* $\boldsymbol{\pi}$ *is a* ***cycle*** *if the starting and finishing states are the same, and we say that a path* contains *a cycle if the same state appears more than once in the path. An FSA for which some* $\boldsymbol{\pi} \in \Pi(\mathcal{A})$ *contains a cycle is called* ***cyclic****.*

Example 3.1.5. $\left(1 \xrightarrow{a} 2, 2 \xrightarrow{c} 4, 4 \xrightarrow{a} 6\right)$ *is an example of a path in the first FSA from Fig. 3.2. An example of a cycle in it would be* $2 \xrightarrow{b} 2$*.*

3.1.3 String Recognition

The notion of a path allows us to introduce a more rigorous definition of what recognizing a string means.

Definition 3.1.5 (Recognized string)**.** *Let* $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ *be a FSA. Let* $\boldsymbol{y} \in \Sigma^*$ *be an arbitrary string. We can define the set of* ***accepting paths*** *for the string* $\boldsymbol{y}$ *as the set* $\Pi(I, \boldsymbol{y}, F)$*. Hence, a string* $\boldsymbol{y} \in \Sigma^*$ *is* ***recognized*** *(or* ***accepted****) by an automaton* $\mathcal{A}$ *if and only if the cardinality of the set of accepting paths is strictly positive, i.e.*

$$|\Pi(I, \boldsymbol{y}, F)| \ > 0 \tag{3.1}$$

[5] Notice we use the Kleene closure on the set δ here. It thus represents any sequence of transitions $\in \delta$

In plain English, the string is accepted by the automaton if and only if there exists at least one path between an initial and a final state which yields the string $\boldsymbol{y}$.

Example 3.1.6. *A few examples of strings accepted by the $\mathcal{A}_1$ in Fig. 3.2a include $bba, bca, aca, abca, abbca, abbbca \ldots$. In fact, due to the self-loop at state 2, the symbol b can appear an arbitrary number of times at position 2 in the accepted string $abca$. Notice that, starting from the state 1 and following the transitions dictated by any of the accepted strings, we always end up in the only final state, state 6.*

In particular, the string "abbca" is accepted with the following set of transitions in $\mathcal{A}_1$:

$$1 \xrightarrow{a} 2, 2 \xrightarrow{b} 2, 2 \xrightarrow{b} 2, 2 \xrightarrow{c} 4, 4 \xrightarrow{a} 6.$$

Example 3.1.7. *The language described by the FSA in Fig. 3.2a could be represented with the formal power series r as $r(a) = \boldsymbol{0}, r(b) = \boldsymbol{0}, r(c) = \boldsymbol{0}, \ldots, r(aca) = \boldsymbol{1}, \ldots, r(abca) = \boldsymbol{1}, \ldots, r(bba) = \boldsymbol{1}, \ldots, r(bca) = \boldsymbol{1}$. As you can see, we are not evaluating the series at any value, we are just using it to represent the strings accepted by the FSA! This is the crucial difference to series such as the one in Example 2.3.2. This notation will become especially useful once we extend FSAs to weighted FSAs (WFSAs) in the next section.*

This leads us to the notion of ambiguity of an FSA.

Definition 3.1.6. *An FSA $\mathcal{A} \stackrel{\text{def}}{=} (\Sigma, Q, I, F, \delta)$ is **unambiguous** if, for every string $\boldsymbol{y} \in \Sigma^*$, there is* at most one *accepting path for that $\boldsymbol{y}$. Otherwise, we say the FSA is **ambiguous**.*

A common point of confusion is the difference between deterministic and unambiguous machines. Determinism implies unambiguity, however, the converse is not true. As an example, consider the machine in Fig. 3.5 that is non-deterministic, but unambiguous.

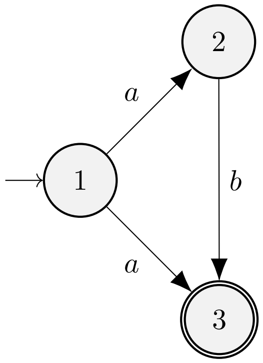


Figure 3.5: A FSA which is non-deterministic but unambiguous.

Example 3.1.8. *The machine shown in Fig. 3.5 accepts a and ab unambiguously but is non-deterministic because we have more than one possible transition from state 1.*

The concept of ambiguity is important and applicable to all of the formalisms we will discuss during this course and we will encounter it often, especially in the context of parsing.

3.2 Weighted Finite-state Automata

In this section we introduce weighted finite-state automata (WFSA), a formalism that will be the primary subject of our discussion over the first 1/3 of the course. We will start with a formal definition WFSA and then move on to a number of applications.

Definition 3.2.1 (Weighted Finite-State Automaton)**.** *A* ***weighted finite-state automaton*** $\mathcal{A}$ *over a semiring* $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ *is a 5-tuple* $(\Sigma, Q, \delta, \lambda, \rho)$ *where*

- Σ *is a finite alphabet;*
- Q *is a finite set of states;*
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \mathbb{K} \times Q$ *a finite multi-set of transitions;*
- $\lambda \colon Q \to \mathbb{K}$ *a weighting function assigning states their initial values;*
- $\rho \colon Q \to \mathbb{K}$ *a weighting function assigning states their final values.*

Notice that we omit the initial and final state sets from the definition of WFSAs. Those can implicitly be specified by the states given non-zero initial or final weights by the λ and ρ functions, i.e., $I = \{q \in Q \mid \lambda(q) \neq 0\}$ and $F = \{q \in Q \mid \rho(q) \neq 0\}$. We might refer to them in the text later for notational convenience and clarity of exposition. This is the definition used in rayuela and the sets I and F are computed on the fly. In the course notes, however, we may still refer to the sets I and F from time to time as it makes the exposition of some algorithms simpler as we may directly refer to the sets I and F by name.

As should be clear, our definition of a WFSA provides a generalization of the FSA where the transitions can be weighted with weights from the semiring. Any unweighted finite state machine can be represented with a weighted one over the boolean semiring. Again, notice that WFSAs over the boolean semiring correspond to characteristic formal power series defined above.

As in the unweighted case, we will often use the more suggestive notation $q_1 \xrightarrow{a/w} q_2$ for the transition $(q_1, a, w, q_2) \in \delta$. Again, we could include ϕ, σ, and ρ in the definition of δ as well.

You can find several WFSAs over different semirings (real, string, and rational) in Fig. 3.6 that correspond to weighted versions of the unweighted FSA presented in Fig. 3.2a. As you can see, the only thing that changed is the fact that the transitions now include weights, which are no longer binary. We write the weights on the edges after the output symbol, separated by a "/". Additionally, the initial and final states now have weights assigned to them as well, where the weights are separated from the state name by the same separator.

3.2.1 Paths and Path Weights

Paths in weighted finite-state automata are a simple extension of those we defined for unweighted machines: they, naturally, additionally include the transition weights.

Definition 3.2.2 (Weighted Path)**.** *A* ***weighted path*** $\boldsymbol{\pi}$ *is an element of* δ^* *with* consecutive transitions*, meaning that it is of the form* $\left(q_1 \xrightarrow{\bullet/\bullet} q_2, q_2 \xrightarrow{\bullet/\bullet} q_3 \ldots q_{n-1} \xrightarrow{\bullet/\bullet} q_n\right)$*, where* $\bullet$ *is a placeholder.*

Notice that the only difference from the unweighted definition of a path lies in the presence of weights in the transitions. Hence, all the related notations and definitions can be trivially extended to the weighted case.

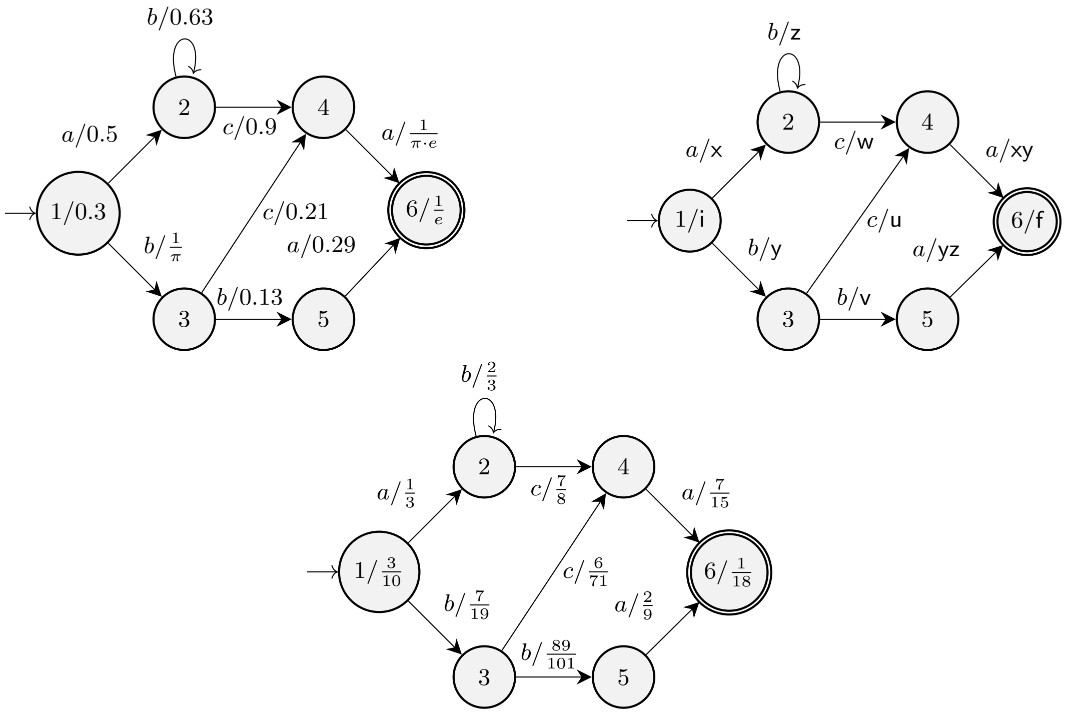


Figure 3.6: The WFSA corresponding to the FSA from Fig. 3.2a over different semirings. You can find the implementation in the Jupyter notebook in rayuela.

Example 3.2.1. $\left(1 \xrightarrow{b/\frac{1}{\pi}} 3, 3 \xrightarrow{b/0.13} 5, 5 \xrightarrow{a/0.29} 6\right)$ *is an example of a path in the first WFSA from Fig. 3.6. An example of a cycle in it would be* $2 \xrightarrow{b/0.63} 2$.

One of the most important questions when talking about weighted formalisms like weighted finite-state automata is how to combine weights of atomic units like transitions into weights of complete *structures*. We begin by multiplicatively combining the weights of individual transitions in a path into the weights of the full path:

Definition 3.2.3 (Path Weight)**.** *The* ***inner path weight*** $w(\boldsymbol{\pi})$ *of a path* $\boldsymbol{\pi} = q_1 \xrightarrow{a_1/w_1} q_2, q_2 \xrightarrow{a_2/w_2} q_3, \cdots q_{N-1} \xrightarrow{a_{N-1}/w_{N-1}} q_N$ *is defined as*

$$w_I(\boldsymbol{\pi}) \stackrel{\text{def}}{=} \bigotimes_{n=1}^{N-1} w_n. \tag{3.2}$$

In the edge case $|\boldsymbol{\pi}| = 0$*, we define the inner path weight to be* $w_I(\boldsymbol{\pi}) \stackrel{\text{def}}{=} \boldsymbol{1}$*. Let* $p(\boldsymbol{\pi})$ *and* $n(\boldsymbol{\pi})$ *denote the origin and the destination states of path* $\boldsymbol{\pi}$*, respectively. The* ***(full) path weight*** *of the path* $\boldsymbol{\pi}$ *is then defined as*

$$w(\boldsymbol{\pi}) \stackrel{\text{def}}{=} \lambda(p(\boldsymbol{\pi})) \otimes w_I(\boldsymbol{\pi}) \otimes \rho(n(\boldsymbol{\pi})). \tag{3.3}$$

A path $\boldsymbol{\pi}$ *is called* ***accepting*** *or* ***successful*** *if* $w(\boldsymbol{\pi}) \neq \boldsymbol{0}$*.*

The inner path weight is therefore the product of the weights of the transitions on the path, while the (full) path weight is the product of the transition weights as well as the initial and final weights of the starting and ending states of the path, respectively.

Example 3.2.2. *Consider the path presented in Example 3.2.1. In this case, the inner path weight would correspond to*

$$w_I\left(\boldsymbol{\pi}\right) = \frac{1}{\pi} \cdot 0.13 \cdot 0.29$$

while the weight path would be

$$w\left(\boldsymbol{\pi}\right) = 0.3 \cdot w_I \cdot \frac{1}{e} = 0.3 \cdot \left(\frac{1}{\pi} \cdot 0.13 \cdot 0.29\right) \cdot \frac{1}{e}$$

3.2.2 String Weights

When we introduced unweighted finite-state automata, we defined the important concept of accepting a *string*. We can then use the defined path weights to generalize these concepts to the very natural quantity of the weight assigned by a WFSA to a string, i.e., it's **acceptance weight**, or **string sum**. Weighted regular languages arise naturally from weighted finite state automata. In contrast to FSA, which map every string to a boolean value (true, if the string is accepted and false otherwise), weighted FSA map strings to weights in $\mathbb{K}$.

Definition 3.2.4 (String sum)**.** *The* ***string sum*** *or string weight of* $\boldsymbol{y}$ *under an automaton* $\mathcal{A}$ *is defined as*

$$\mathcal{A}\left(\boldsymbol{y}\right) \stackrel{\text{def}}{=} \bigoplus_{\boldsymbol{\pi} \in \Pi(\boldsymbol{y})} \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes w_I\left(\boldsymbol{\pi}\right) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right). \tag{3.4}$$

It is interesting to observe that, once again, the unweighted case nicely ties back to the weighted definition. In fact, Eq. (3.1) can be seen as the string sum computed in the boolean semiring, where the definition of $\hat{Q}_{\boldsymbol{y}}$ implies the existence of at least one path $\boldsymbol{\pi} : \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes w_{\mathrm{I}}\left(\boldsymbol{\pi}\right) \neq \mathbf{0}$ and its intersection with F implies that $\rho\left(n\left(\boldsymbol{\pi}\right)\right) \neq \mathbf{0}$.

3.2.3 A Few More Definitions

We finish the section by putting down a few more definitions which will come useful throughout our treatment of WFSAs.

Definition 3.2.5. *We mark the set of* ***outgoing arcs*** *from a state* q *as*

$$\mathcal{E}_{\mathcal{A}}\left(q\right) \stackrel{\text{def}}{=} \left\{q \xrightarrow{a/w} t \mid q \xrightarrow{a/w} t \in \delta\right\}, \tag{3.5}$$

where the subscript may be omitted if the referred automaton is clear from the context. More generally, if $\mathcal{Q} \subseteq Q$*,*

$$\mathcal{E}\left(\mathcal{Q}\right) \stackrel{\text{def}}{=} \bigcup_{q \in \mathcal{Q}} \mathcal{E}\left(q\right). \tag{3.6}$$

Analogously, we define the set of ***incoming arcs*** *into a state* q *as*

$$\mathcal{E}^{-1}{}_{\mathcal{A}}\left(q\right) \stackrel{\text{def}}{=} \left\{p \xrightarrow{a/w} q \mid p \xrightarrow{a/w} q \in \delta\right\} \tag{3.7}$$

with the optional subscript and analogous extension to a set of states $\mathcal{Q}$.

Definition 3.2.6 (Reverse automaton)**.** *The reverse WFSA of a WFSA* $\mathcal{A}$*, marked by* $\overleftarrow{\mathcal{A}}$*, is a WFSA obtained by reversing the direction of all transitions in* $\mathcal{A}$ *and swapping the initial and final states (with the initial weights becoming final and vice-versa).*

Definition 3.2.7 (Accessibility)**.** *A state $q \in Q$ is* ***accessible*** *if there exists a path from I to q.*

Definition 3.2.8 (Co-accessibility)**.** *A state $q \in Q$ is* ***co-accessible*** *if there exists a path from q to F.*

Definition 3.2.9 (Trim)**.** *States that are not both accessible and co-accessible are called* ***useless*** *states. An automaton without useless states and without transitions of weight* ***0*** *is called* ***trim****.*

Lastly, we define a few important quantities which will come up repeatedly throughout our exploration of weighted automata. These are the **forward** and **backward** weights. We will talk more about them in §3.6.2.

Definition 3.2.10. *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA over the semiring $\mathcal{W}$. Let $\boldsymbol{y} \in \Sigma^*$ be a prefix.*[6]

The ***forward weight*** *of the prefix $\boldsymbol{y}$ is defined as*

$$\boldsymbol{\alpha}_{\mathcal{A}}(\boldsymbol{y}) \overset{\text{def}}{=} \bigoplus_{\substack{\boldsymbol{\pi} \in \Pi(\mathcal{A}), \\ \mathrm{s}(\boldsymbol{\pi}) = \boldsymbol{y}}} \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes w_I\left(\boldsymbol{\pi}\right). \tag{3.8}$$

The ***forward weight*** *of the prefix $\boldsymbol{y}$* to the state q *is defined as*

$$\boldsymbol{\alpha}_{\mathcal{A}}(\boldsymbol{y}, q) \overset{\text{def}}{=} \bigoplus_{\substack{\boldsymbol{\pi} \in \Pi(\mathcal{A}), \\ \mathrm{s}(\boldsymbol{\pi}) = \boldsymbol{y}, n(\boldsymbol{\pi}) = q}} \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes w_I\left(\boldsymbol{\pi}\right). \tag{3.9}$$

Lastly, the ***forward weight*** *of the* state q *is defined as*

$$\boldsymbol{\alpha}_{\mathcal{A}}(q) \overset{\text{def}}{=} \bigoplus_{\substack{\boldsymbol{\pi} \in \Pi(\mathcal{A}), \\ n(\boldsymbol{\pi}) = q}} \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes w_I\left(\boldsymbol{\pi}\right). \tag{3.10}$$

Definition 3.2.11. *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA over the semiring $\mathcal{W}$. Let $\boldsymbol{y} \in \Sigma^*$ be a suffix.*[7]

The ***backward weight*** *of the suffix $\boldsymbol{y}$ is defined as*

$$\boldsymbol{\beta}_{\mathcal{A}}(\boldsymbol{y}) \overset{\text{def}}{=} \bigoplus_{\substack{\boldsymbol{\pi} \in \Pi(\mathcal{A}), \\ \mathrm{s}(\boldsymbol{\pi}) = \boldsymbol{y}}} w_I\left(\boldsymbol{\pi}\right) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right). \tag{3.11}$$

The ***backward weight*** *of the suffix $\boldsymbol{y}$* to the state q *is defined as*

$$\boldsymbol{\beta}_{\mathcal{A}}(\boldsymbol{y}, q) \overset{\text{def}}{=} \bigoplus_{\substack{\boldsymbol{\pi} \in \Pi(\mathcal{A}), \\ \mathrm{s}(\boldsymbol{\pi}) = \boldsymbol{y}, p(\boldsymbol{\pi}) = q}} w_I\left(\boldsymbol{\pi}\right) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) \tag{3.12}$$

Lastly, the ***backward weight*** *of the* state q *is defined as*

$$\boldsymbol{\beta}_{\mathcal{A}}(q) \overset{\text{def}}{=} \bigoplus_{\substack{\boldsymbol{\pi} \in \Pi(\mathcal{A}), \\ p(\boldsymbol{\pi}) = q}} w_I\left(\boldsymbol{\pi}\right) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) \tag{3.13}$$

This concludes our definition of WFSAs.

[6] A prefix is not necessarily a string in the language, but is, as the name suggests, a prefix of a string in the language.

[7] Similarly, a suffix is not necessarily a string in the language, but is, as the name suggests, a suffix of a string in the language.

3.3 (Weighted) Regular Languages

Leveraging the definition of the acceptance of a string (cf. Def. 3.1.5), we can define the language accepted by an automaton as

Definition 3.3.1. *The set $L(\mathcal{A}) \subseteq \Sigma^*$ of all strings $\boldsymbol{y} \in \Sigma^*$ that a finite-state automaton $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ can accept, commonly referred to as the **language accepted by** $\mathcal{A}$, is then naturally defined as*

$$L(\mathcal{A}) = \{\boldsymbol{y} \in \Sigma^* \mid |\Pi(I, \boldsymbol{y}, F)| \ > 0\} \tag{3.14}$$

The set of languages that finite-state automata can recognize are known as the **class of regular languages**.

Definition 3.3.2. *A language $L \subseteq \Sigma^*$ is **regular** if and only if it can be recognized by an unweighted finite-state automaton.*

We defined regular languages in terms of their acceptance by *unweighted* automata. We say that a language is accepted by a WFSA if it is accepted by the unweighted version of the automaton obtained by ignoring the weights (and removing the **0**–weighted ones)—notice that this corresponds to the support of the formal power series encoded by the WFSA. We denote the language accepted by the automaton $\mathcal{A}$ by $L(\mathcal{A})$.

Definition 3.3.3 (Regular Weighted Language)**.** *A* weighted *language L is **regular** if and only if it can be recognized by a weighted finite-state automaton.*

3.4 Applications of Weighted Finite-state Automata

Example 3.4.1 (N-gram models)**.** *The first application we will look at comes from natural language processing. We will consider the task of* ***language modeling****, i.e., modeling the probability of a sequence of words from some vocabulary. Formally, let* $\mathbf{Y} = (y_1, \dots y_M) \in \mathcal{Y}^M$ *represent some sequence of words of length* M *and let* y_n *denote the word at position* n*. We want to model the joint probability* $p(\mathbf{Y})$*, which we can decompose into the product of conditional ones as* $p(\mathbf{Y}) = \prod_{n=1}^{M} p(y_n \mid y_{<n})$*. This means that it is sufficient to model the individual conditional probabilities* $p(y_n \mid y_1 \dots y_{n-1}) = p(y_n \mid y_{<n})$*. These can be seen as "modeling the probabilities of next occurring words given the ones which have been observed so far". Additionally, we introduce two special tokens,* BOS *(standing for* beginning of sentence*) and* EOS *(*end of sentence*), to be able to handle the edge cases at the beginning and end of a sentence. They will always represent the first and last token in a sentence, i.e.,* $y_1 = \mathsf{BOS}$ *and* $y_M = \mathsf{EOS}$*. They allow us to model the probability of different sequences starting* $(p(y \mid \mathsf{BOS}))$ *or ending* $(p(\mathsf{EOS} \mid y_{<M}))$ *the sentence.*

Since the space of possible sequences of words is a (large) discrete space, the probability distribution will be discrete as well. However, notice that the probability distribution $p(y_n \mid y_{<n})$ *depends on the value of* $y_{<n}$*, which can be any of* $|\mathcal{Y}|^{n-1}$ *combinations of words, where* $|\mathcal{Y}|$ *is the size of the vocabulary. If we represent these probabilities in discrete tables, the exponential growth of the number of possible combinations quickly makes the situation unmanageable. This is why we have to make some assumptions on* $p(\mathbf{Y})$ *to end up with a manageable task.*

One of the simplest restrictions on the conditional distributions is the N***-gram assumption****. It says that the probability of the next occurring word only depends on the previous* N *words in the text, i.e.:*

$$p(y_n \mid y_{<n}) = p(y_n \mid y_{n-N+1} \dots y_{n-1}). \tag{3.15}$$

This caps the number of possible probability distributions we have to keep track of at $|\mathcal{Y}|^N$*.*[8] *Some care needs to be taken for cases when* $n < N$*, which is where special tokens such as* BOS *(beginning of sentence) come in.*

An especially simple case of the N*-gram model is the so-called* ***bi-gram model****, where the probabilities of the next word only depend on the previous one, i.e.* $p(y_n \mid y_{<n}) = p(y_n \mid y_{n-1})$*. This is also called the* ***Markov assumption****, which we will take a closer look at in the next example.*

Let us look at a specific example of a simple bi-gram model. Suppose our vocabulary consists of the words formal*,* $\mathsf{language}$*, and* theory*, thus,* $|\mathcal{Y}| = 3$*. To specify the* N*-gram model, we have to define the conditional probabilities* $p(y_j \mid y_i)$ $\forall y_i, y_j \in \mathcal{Y} \cup \{\mathsf{BOS}, \mathsf{EOS}\}$*. In the case of bi-grams, we can represent those in a table like the following, where the entry at position* i, j *represents the probability* $p(y_j \mid y_i)$*:*

	BOS	*formal*	*language*	*theory*	*EOS*
BOS	0.0	0.4	0.2	0.2	0.2
formal	0.0	0.1	0.4	0.2	0.3
language	0.0	0.1	0.1	0.4	0.4
theory	0.0	0.2	0.2	0.1	0.5
EOS	1.0	0.0	0.0	0.0	0.0

[8] How many parameters do we have to keep track of, though?

Under our model, the probability of the sentence "formal language theory" would be

$$p\left(\textsf{formal} \mid \textsf{BOS}\right) p\left(\textsf{language} \mid \textsf{formal}\right) p\left(\textsf{theory} \mid \textsf{language}\right) p\left(\textsf{EOS} \mid \textsf{theory}\right) = \\ = 0.4 \cdot 0.4 \cdot 0.4 \cdot 0.5 = 0.032$$

while the probability of the sentence "formal formal formal" would be

$$p\left(\textsf{formal} \mid \textsf{BOS}\right) p\left(\textsf{formal} \mid \textsf{formal}\right) p\left(\textsf{formal} \mid \textsf{formal}\right) p\left(\textsf{EOS} \mid \textsf{formal}\right) = \\ = 0.4 \cdot 0.1 \cdot 0.1 \cdot 0.3 = 0.0012.$$

Note that the probabilities in the above table are made up and not completely reasonable. A real N-gram model would not allow for probabilities of exactly 0 to avoid pathological behavior.

So how can we represent an N-gram language model as a WFSA? The idea is to construct a WFSA whose states will represent all possible sequences of words of length N. The transitions between the states q_1 and q_2 will correspond to the possible transitions between the N-grams which those represent. This means that the only possible (positively-weighted) transitions will be between the N-grams which can follow each other, i.e. $y_{-N}y_{-(N-1)}\cdots y_{-1}$ and $y_{-(N-1)}\cdots y_{-1}y_0$ for some $y_{-N}, y_0 \in \mathcal{Y}$, with the weight probability being the probability of observing the "new" word y_0 in the second N-gram given the starting N-gram $y_{-N}y_{-(N-1)}\cdots y_{-1}$. Formally, we can map any N-gram model into a WFSA $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ by constructing the WFSA like this:

- $\Sigma = \mathcal{Y} \cup \{\textsf{BOS}, \textsf{EOS}\}$
- $Q = \bigcup_{n=0}^{N} \{\textsf{BOS}\}^{N-n} \times \mathcal{Y}^n$
- $I = \left\{ \underbrace{\textsf{BOS} \ldots \textsf{BOS}}_{N\ times} \right\}$
- $F = Q$
- $\delta = \left\{ \left(y_{-N}y_{-(N-1)}\cdots y_{-1}, y_0, p\left(y_0 \mid y_{-(N-1)}\cdots y_{-1}\right), y_{-(N-1)}\cdots y_{-1}y_0 \right) \right.$
  $\left. \textit{for}\ \left(y_{-(N-1)}\cdots y_{-1}\right) \in U \right\}\ \textit{where}\ U = \bigcup_{n=0}^{N-1} \{\textsf{BOS}\}^{N-1-n} \times \mathcal{Y}^n; y_0, y_{-N} \in \mathcal{Y}$
- $\lambda : \underbrace{\textsf{BOS} \ldots \textsf{BOS}}_{N\ times} \mapsto \boldsymbol{1}$
- $\rho : y_{-(N-1)}\cdots y_{-2}y_{-1}\textsf{EOS} \mapsto \boldsymbol{1}$

Notice that the definition of the state space is not completely straight-forward. We define it as $Q = \bigcup_{n=0}^{N} \{\textsf{BOS}\}^{N-n} \times \mathcal{Y}^n$, which ensures that we can handle tokens appearing at positions $n < N$ by padding the beginning with BOS*. Since* EOS *finishes a sentence, all N-gramns ending with* EOS *are assigned the final weight of **1**, while the transition probability $p\left(\textsf{EOS} \mid y_{-(N-1)}\cdots y_{-1}\right)$ models the probability of the sequence $y_{-(N-1)}\cdots y_{-1}$ actually ending a sentence.*

We thus have a recipe for representing any N-gram language model with a WFSA. We will develop many theoretical results and practical algorithms on those in this course. This mapping then allows us to use all the machinery we will develop for WFSA on N-gram language models, making it just a special case of the very general framework.

The WFSA corresponding to the bi-gram model from above is presented in Fig. 3.7.

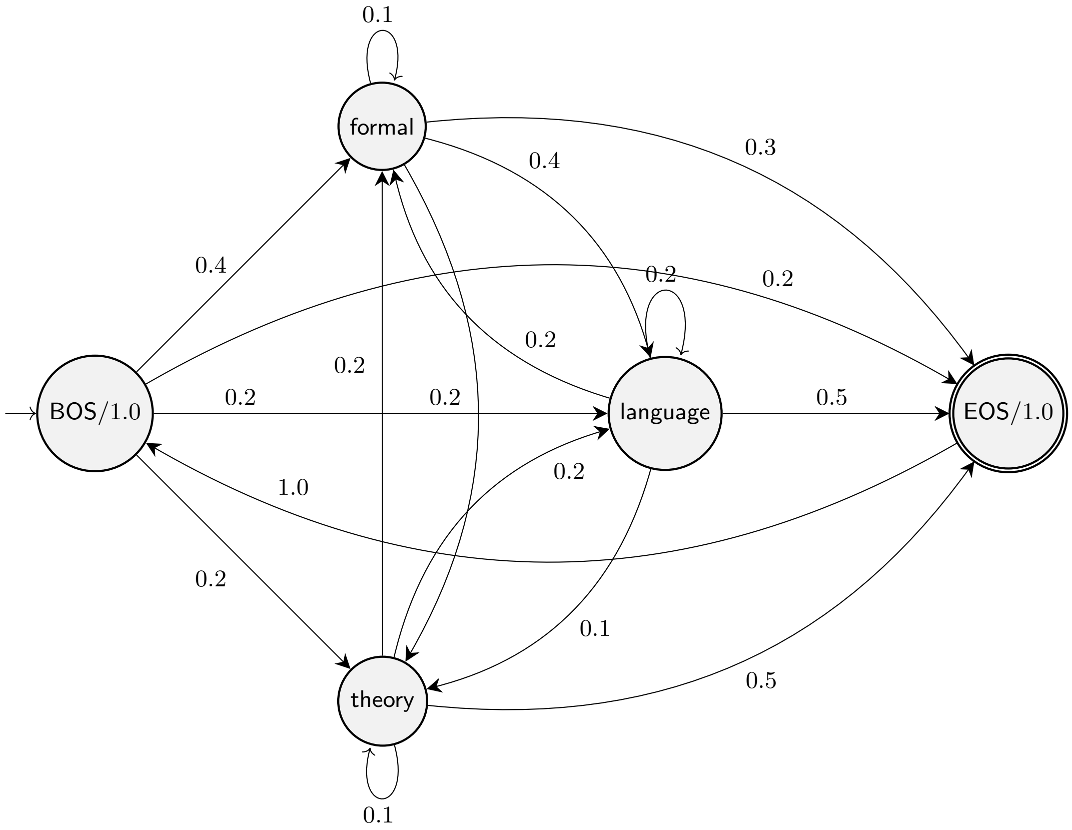


Figure 3.7: The WFSA corresponding to the simple bi-gram model introduced above. To avoid a cluttered figure, the edge labels are left out—all the edges into node x have the label x. You can find the implementation in the Jupyter notebook of Chapter 1.

Example 3.4.2 (Hidden Markov Model)**.** *A* ***hidden Markov model (HMM)*** *is a common statistical model coming up in fields like physics, chemistry, bioinformatics, and many more. In real world situations, we often observe a sequence of* observable states *(variables) x_n, but what we are interested in are the unobservable,* hidden*, states y_n which we suppose generated the observables (or are at least associated with them). An example would be part-of-speech tags in natural language processing, where we can observe the words in a sentence, while the tags associated with them are unobserved. HMMs are a probabilistic tool which offers a framework for modeling such settings and, among other things, inferring the hidden states. Let us have a look at how we can concretely use the introduced WFSA to represent HMMs. In this section, rather than being as general as possible, we are focusing on a simplified, but commonly used, version of HMMs, since the goal is only to make the connection to WFSA.*

HMMs are ***generative models****, meaning that they model the joint probability of both the observable as well as the hidden random variables. We will denote the sequence of observable random variables of length N with $\mathbf{X}$ and the sequence of hidden* of the same length *with $\mathbf{Y}$. With a hidden Markov model, we model the* ***joint probability distribution*** *as follows:*

$$p\left(\mathbf{X}, \mathbf{Y}\right) = \prod_{n=1}^{N} p\left(y_n \mid y_{n-1}\right) p\left(x_n \mid y_n\right). \tag{3.16}$$

That is, the model imposes constraints on the conditional dependencies of the random variables, *namely:*

(i) the distribution of the n-th observable random variable x_n only depends on the value of the n-th hidden variable y_n (the one that "caused" it):

$$p\left(x_n \mid y_1 \ldots y_n, \{x_j,\ j \neq i\}\right) = p\left(x_n \mid y_n\right), \tag{3.17}$$

*(ii) the **Markov assumption** on the hidden variables:*

$$p\left(\mathbf{Y}\right) = \prod_{n=1}^{N} p\left(y_n \mid y_{n-1}\right), \tag{3.18}$$

*meaning that the hidden random variables y_n form a **Markov chain**.*

We can therefore look at an HMM as an augmentation of the normal Markov chain of the hidden states y_n using the observations x_n as a "noisy" proxy for them.

We can see that the model decomposes the joint probability into the product of many smaller local terms, which are themselves (normalized) probability distributions. *Such decomposition of the joint probability makes HMMs **locally normalized models**. This is an important point which distinguishes them from **globally normalized models** which we will introduce in the next example. This distinction is another concept we will encounter repeatedly in the course.*

*To make the connection to WFSA clearer (and as is often done in practice) we assume the state space of a Markov chain is discrete and identical for all the variables in the sequence, i.e., $y_n \in \{1, \ldots, K\} = \mathcal{Y}$ for some $K \in \mathbb{N}$. Additionally, we assume the chain is **homogeneous**, meaning that the conditional probabilities are the same for all n. In that case, a Markov chain is fully defined in terms of:*

- *an **initial distribution** Π over y_1*
- *a **transition probability distribution** $p\left(y_n = j \mid y_{n-1} = k\right)$, which is identical $\forall n$. It can be concisely represented as a $K \times K$ matrix $\mathbf{T}$ with $\mathbf{T}_{kj} = p\left(y_n = j \mid y_{n-1} = k\right)$.*

*The Hidden Markov Model extends Markov chains by adding the (discrete) observable variables $x_n \in \{1, \ldots, L\} = \mathcal{X}$, which depend on the hidden variables in terms of the **emission probabilities** $p\left(x_n = j \mid y_n = k\right)$. These can also be represented as a $K \times L$ matrix $\mathbf{E}$ as $\mathbf{E}_{kj} = p\left(x_n = j \mid y_n = k\right)$.*

A fun example of an HMM comes from Jason Eisner's tutorial (Eisner, 2002), summarized from Jurafsky and Martin (2022). Suppose we wanted to predict the outside temperature (which can be either HOT or COLD) using only the information about how many ice creams Jason ate on a particular day. Suppose he is mindful of his health and only eats up to 3 ice creams per day, depending on the weather. We could represent his indulgence habits with the following probability table:

	1	2	3
COLD	0.5	0.4	0.1
HOT	0.2	0.4	0.4

meaning, for example, that the probability that Jason eats 3 ice creams on a cold day is 0.1. *Based on our knowledge of the weather, we also construct the following weather model, which represents how warm and cold days follow each other:*

	COLD	HOT
COLD	0.5	0.5
HOT	0.6	0.4

meaning that a hot and a cold day follow a cold one with equal probability and that a day is slightly more likely to be cold than hot if the previous one was hot as well. Suppose that a hot and a cold day is also equally likely at the beginning. This represents an HMM with $\mathcal{Y} = \{\textsf{COLD}, \textsf{HOT}\}$ and $\mathcal{X} = \{1, 2, 3\}$. Transitions and emissions are given with the probability tables above.

We can also represent HMMs graphically in two ways. The first one is shown in Fig. 3.8 and represents the HMM as a **directed graphical model**. The unshaded nodes represent the individual hidden random variables, which themselves can take any of the K values. The shaded nodes represent observable random variables, which can take L different values. While a thorough interpretation of the figure would require an in-depth look into graphical models, it is enough to realize 2 things. Firstly, the latent part of the picture shows the Markov chain of hidden variables — the linear chain of nodes represents the Markov nature of the model (arrows, which represent ”influences”, only connect subsequent y_n's). Secondly, the arrows connecting y_n's with x_n's represent the emission relationships and show that the distribution of x_n only depends on the value of the corresponding y_n (since the only incoming arc into any x_n is coming from y_n).

The second representation can be seen in Fig. 3.9. In it, the nodes of the latent space represent the values which those can take (meaning that, at any point, only one latent variable in a column will be “active”). Note that this figure corresponds to the transition weights under a specific observed sequence $\mathbf{Y}$. The fact that it corresponds to a specific observed sequence is apparent from the single edges connecting the layers—we only use the weights dictated by the observed y_n (through the emission probabilities). This graphical representation will also allow us to make the connection to WFSA shortly.

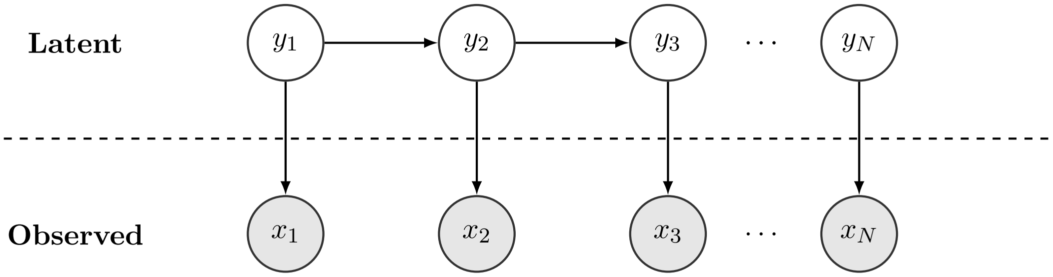


Figure 3.8: A schematic view of an HMM as a **directed graphical model**.

HMMs allow us to encode our belief of how the world behaves in terms of the influences of its different parts on each other. For now, we assume that the transition and emission probabilities are given, but in practice, they are learned. With the weights set, the unknown quantities we might be interested in include the **marginal probability of the observed sequence**:

$$p\left(\mathbf{X}\right) = \sum_{\mathbf{Y} \in \mathcal{Y}^N} p\left(\mathbf{X}, \mathbf{Y}\right) = \sum_{\mathbf{Y} \in \mathcal{Y}^N} \prod_{n=1}^{N} p\left(y_n \mid y_{n-1}\right) p\left(x_n \mid y_n\right), \tag{3.19}$$

or the **most probable hidden sequence $\mathbf{Y}$ given the observed variables**:

$$\mathbf{Y}^*\left(\mathbf{X}\right) = \operatorname*{argmax}_{\mathbf{Y} \in \mathcal{Y}^N} \left\{p\left(\mathbf{Y} \mid \mathbf{X}\right)\right\} = \operatorname*{argmax}_{\mathbf{Y} \in \mathcal{Y}^N} \prod_{n=1}^{N} p\left(y_n \mid y_{n-1}\right) p\left(x_n \mid y_n\right). \tag{3.20}$$

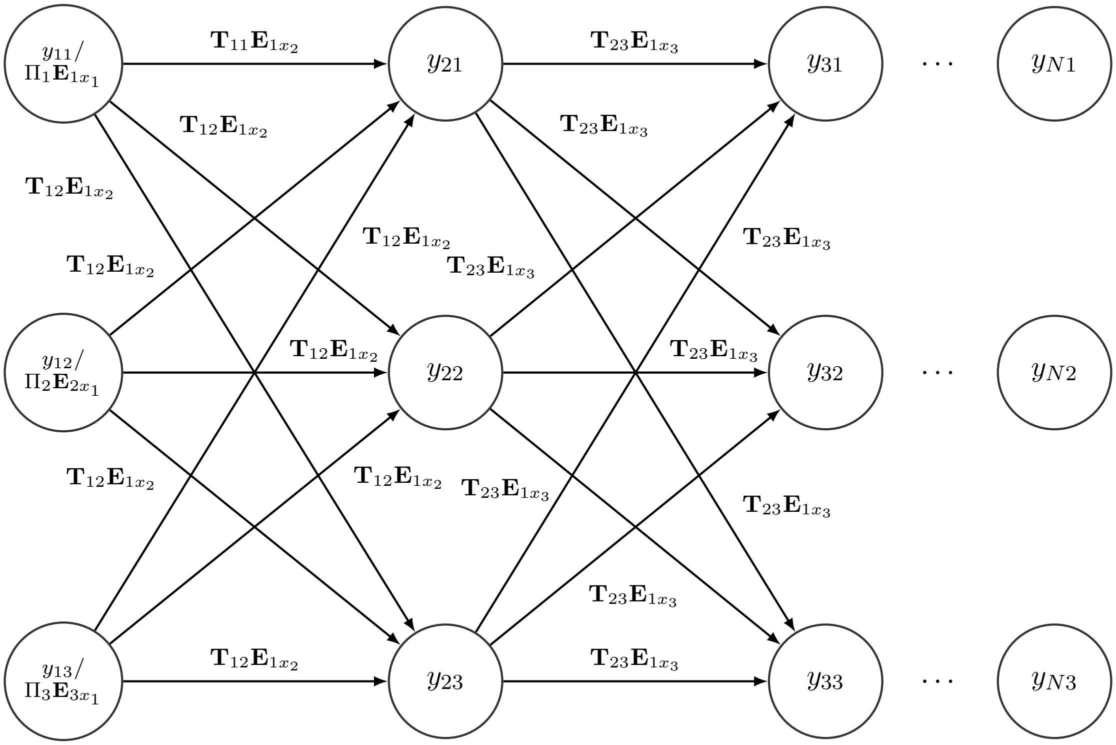

$y_{11}/$ $\Pi_1\mathbf{E}_{1x_1}$
$\mathbf{T}_{11}\mathbf{E}_{1x_2}$
y_{21}
$\mathbf{T}_{23}\mathbf{E}_{1x_3}$
y_{31}
$\cdots$
y_{N1}
$\mathbf{T}_{12}\mathbf{E}_{1x_2}$
$\mathbf{T}_{23}\mathbf{E}_{1x_3}$
$\mathbf{T}_{12}\mathbf{E}_{1x_2}$
$\mathbf{T}_{12}\mathbf{E}_{1x_2}$
$\mathbf{T}_{23}\mathbf{E}_{1x_3}$
$\mathbf{T}_{12}\mathbf{E}_{1x_2}$
$\mathbf{T}_{23}\mathbf{E}_{1x_3}$
$y_{12}/$ $\Pi_2\mathbf{E}_{2x_1}$
$\mathbf{T}_{12}\mathbf{E}_{1x_2}$
y_{22}
$\mathbf{T}_{23}\mathbf{E}_{1x_3}$
y_{32}
$\cdots$
y_{N2}
$\mathbf{T}_{12}\mathbf{E}_{1x_2}$
$\mathbf{T}_{12}\mathbf{E}_{1x_2}$
$\mathbf{T}_{23}\mathbf{E}_{1x_3}$
$\mathbf{T}_{23}\mathbf{E}_{1x_3}$
$\mathbf{T}_{23}\mathbf{E}_{1x_3}$
$y_{13}/$ $\Pi_3\mathbf{E}_{3x_1}$
$\mathbf{T}_{12}\mathbf{E}_{1x_2}$
y_{23}
$\mathbf{T}_{23}\mathbf{E}_{1x_3}$
y_{33}
$\cdots$
y_{N3}

Figure 3.9: An expanded view of an HMM as a lattice with the size of the hidden state space $K = 3$ *for a specific observed sequence* $\mathbf{Y}$. The observed variables x_n determine the transition probabilities between the hidden states via the emission matrix $\mathbf{E}$. Importantly, this is not yet a WFSA representation of an HMM, but it hopefully makes the connection clearer.

Both formulas come from basic rules of probability — to get the marginal probability of the observed sequence, we marginalize out the hidden sequence (which corresponds to a sum in this discrete case), and to get the probability of a hidden sequence given the observed one, we use the Bayes rule (when maximizing it, we also ignore the denominator $p(\mathbf{X})$*, since it is constant and does thus not influence the maximization).*

*As you can see, both expressions have a very similar structure - the (*arg*)*max *operation replaces the* $\sum$ *one in the second equation. Notice that both also have, if computed naïvely, an* exponential number of terms *due to the sum/maximization over* $\mathcal{Y}^N$*, so we will have to be smart if we are ever going to compute them. After we encode an HMM as an instance of a WFSA, we will see that that these two expressions are actually an instance of the same general problem of* path sum *in two different semirings! Therefore, any algorithms we use for those will be applicable for these specific problems.*

This means that we could now, with the HMM defined above, for example, reason about what the weather is using only the information about how much ice cream Jason ate.

To make the connection to WFSA, we can observe that, in the discrete homogeneous case we outlined above, we can think of the hidden state space $\mathcal{Y}$ *as a set of states, and the transitions governed by the conditional probabilities as the transitions through these states. Before defining the mapping formally, let us try to imagine how we might go about it. The HMM contains a separate tuple of random variables* (y_n, x_n) $\forall n \in [N]$*. The WFSA will represent this sequence of variables using* one *set of states, one for each* $y \in \mathcal{Y}$ *(plus one additional initial state), and the transitions will depend on the observations* x*. Each state will correspond to any possible value that a hidden* *variable can take, i.e., $Q = \mathcal{Y}$. The transition function will move through these states based on the observed sequence (= the input string of the WFSA). Thus, the transitions will be labeled with the possible emitted symbols and weighted according to the emission probabilities.*

Formally, we can define a WFSA corresponding to an HMM as follows:

- $\Sigma = \{1, \dots, L\} = \mathcal{X}$
- $Q = \{1, \dots, K\} \cup \{q_0\} = \mathcal{Y} \cup \{q_0\}$
- $I = \{q_0\}$
- $F = Q \setminus \{q_0\}$
- $\delta = \{(q_0, k, \Pi_j \cdot \mathbf{E}_{jk}, j) \text{ for } j \in \mathcal{Y}, k \in \mathcal{X}\} \cup \{(i, k, \mathbf{T}_{ij} \cdot \mathbf{E}_{jk}, j) \text{ for } i, j \in \mathcal{Y}, k \in \mathcal{X}\}$
- $\lambda : q_0 \mapsto \boldsymbol{1}$
- $\rho : q \mapsto \boldsymbol{1} \; \forall q \in Q \setminus \{q_0\}$

To see this in action, we present the ice-cream-based weather prediction HMM as a WFSA in Fig. 3.10.

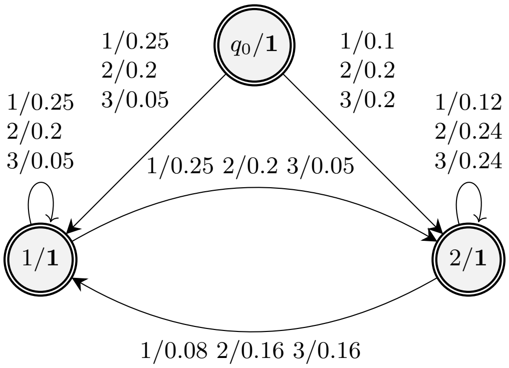


Figure 3.10: The WFSA corresponding to the ice-cream-based weather prediction HMM. You can find the implementation in the Jupyter notebook of Chapter 1.

This allows us to use all the machinery we will develop for WFSA on HMMs. Among other things, it allows us to calculate the quantities $\mathbf{Y}^*$ and $p(\mathbf{X})$ from above with very general and efficient algorithms.

Example 3.4.3 (Conditional Random Field)**.** *The last example we look at is the **conditional random field (CRF)**. You can think of it as a flexible* discriminative *version of a Hidden Markov Model, i.e., one where we are not interested in the joint probability of the observable and hidden variables $p(\mathbf{X}, \mathbf{Y})$, but rather only in $p(\mathbf{Y} \mid \mathbf{X})$.*

*We now no longer require $\mathbf{Y}$ and $\mathbf{X}$ to be of the same length. Furthermore, while $\mathbf{Y}$ still describes some discrete quantity (a structure), $\mathbf{X}$ can now also be continuous. Suppose $\mathbf{X}$ is of length M and $\mathbf{Y}$ of length N. The way we model the distribution $p(\mathbf{Y} \mid \mathbf{X})$ is a bit different to how we do it in HMMs. To get the marginal distribution of the hidden variables given the observable ones (the **posterior**) there, we used the Bayes rule. In CRFs, we model it directly, but, instead of defining it based on the raw values of the observed variables $\mathbf{X}$, we define it with regards to a notion of "compatibility" between $\mathbf{X}$ and any possible $\mathbf{Y}$. This compatibility is modeled through a*

score function *ϕ, which captures how compatible a pair of $\mathbf{X}$ and $\mathbf{Y}$ are, with large (positive) values meaning strong compatibility and small (negative) values meaning no compatibility. This compatibility function could in general take any form, as long as it takes in the pair of observed and hidden variables and returns a scalar, but in the context of CRFs, ϕ takes a specific form. It is defined through a* ***feature function*** *$\mathbf{f}$, which captures which aspects of the observed variables $\mathbf{X}$ influence $\mathbf{Y}$. It maps $\mathbf{X}$ and $\mathbf{Y}$ into a (long) vector, where each dimension corresponds to one aspect of interaction between $\mathbf{X}$ and $\mathbf{Y}$. $\mathbf{f}$ is therefore a function from $\mathcal{X}^M \times \mathcal{Y}^N \to \mathbb{R}^d$. A concrete example will be given shortly. The way the produced features then influence the probability over $\mathbf{Y}$ is determined by the weights $\mathbf{w} \in \mathbb{R}^d$, where the individual contributions are combined* additively. *This results in the score function $\phi_{\mathbf{w}}(\mathbf{X}, \mathbf{Y}) = \mathbf{w}^\top \mathbf{f}(\mathbf{X}, \mathbf{Y})$. This allows for a very general formulation of the conditional probability in a CRF $p(\mathbf{Y} \mid \mathbf{X})$ as:*

$$p(\mathbf{Y} \mid \mathbf{X}) = \frac{1}{Z(\mathbf{X})} \exp\left[\phi(\mathbf{Y}, \mathbf{X})\right] = \frac{1}{Z(\mathbf{X})} \exp\left[\mathbf{w}^\top \mathbf{f}(\mathbf{Y}, \mathbf{X})\right] \tag{3.21}$$

where

$$Z(\mathbf{X}) = \sum_{\mathbf{Y}' \in \mathcal{Y}^N} \exp\left[\phi\left(\mathbf{Y}', \mathbf{X}\right)\right]. \tag{3.22}$$

Notice that the scores are exponentiated. This is done to be able to derive a valid probability distribution based on the them without restricting the range of $\mathbf{w}$. Since the exponential function is monotone, it does not change the compatibility ordering.

We can again see that the expression for $Z(\mathbf{X})$ involves exponentially many terms, which means we will not be able to compute it in reasonable time without resorting to some clever tricks. In fact, what is needed is a further assumption on the form of the feature function $\mathbf{f}$. To avoid having to enumerate all possible combinations the hidden sequence $\mathbf{Y}$ can take, we impose that $\mathbf{f}$ additively decomposes over the dimensions of $\mathbf{Y}$ as:

$$\mathbf{f}(\mathbf{X}, \mathbf{Y}) = \sum_{n=1}^{N} \mathbf{f}(y_n, y_{n-1}, \mathbf{X}). \tag{3.23}$$

CRFs with scoring functions of this form are called ***linear chain CRFs****. We have used an overloaded definition of $\mathbf{f}$ to simplify notation. We can again make use of (an equivalent of) the special BOS symbol for the edge starting case. Notice that the features can still depend on the* entire *$\mathbf{X}$, since it is constant when calculating the normalizing constant $Z(\mathbf{X})$ and thus does not contribute to the complexity. In contrast, we have reduced the complexity of interaction between different parts of $\mathbf{Y}$ to the interaction between two subsequent elements! As we will see, this will allow us, with some clever operations, to calculate $Z(\mathbf{X})$ very efficiently!*

With this restriction, we can now rewrite the conditional probability as:

$$p(\mathbf{Y} \mid \mathbf{X}) = \frac{1}{Z(\mathbf{X})} \exp\left[\sum_{n=1}^{N} \mathbf{w}^\top \mathbf{f}(y_n, y_{n-1}, \mathbf{X})\right] = \frac{1}{Z(\mathbf{X})} \prod_{n=1}^{N} \exp\left[\mathbf{w}^\top \mathbf{f}(y_n, y_{n-1}, \mathbf{X})\right] \tag{3.24}$$

where now

$$Z(\mathbf{X}) = \sum_{\mathbf{Y}' \in \mathcal{Y}^N} \prod_{n=1}^{N} \exp\left[\mathbf{w}^\top \mathbf{f}\left({y_n}', {y_{n-1}}', \mathbf{X}\right)\right]. \tag{3.25}$$

The term for calculating $Z(\mathbf{X})$ still involves exponentially many terms if computed naïvely. However, as we will soon see, we can rearrange the terms to make computations efficient.

Similarly to the conditional probabilities in HMMs, the set of weights $\mathbf{w}$ is learned *according to data.*

Unlike an HMM, which decomposes the joint probability over a set of smaller conditional probabilities, a CRF is a ***globally normalized model****, where the scores of the individual sequences $\mathbf{Y}$ are normalized with the global normalizer $Z(\mathbf{X})$.*

The main tasks when working with a CRF are computing the normalization constant $Z(\mathbf{X})$ as we saw above, and finding the ***maximum scoring sequence*** *$\mathbf{Y}^*$:*

$$\mathbf{Y}^*(\mathbf{X}) = \underset{\mathbf{Y} \in \mathcal{Y}^N}{\operatorname{argmax}} \prod_{n=1}^{N} \exp\left[\mathbf{w}^\top \mathbf{f}(y_n, y_{n-1}, \mathbf{X})\right], \tag{3.26}$$

which also involves exponentially many terms. With the decomposed form of $\mathbf{f}$, the expressions for $Z(\mathbf{X})$ and $\mathbf{Y}^*(\mathbf{X})$ again look very similar and again, they are instances of the same operation on a WFSA defined over different semirings!

The graphical representation of the interactions among the different variables in a CRF is shown in Fig. 3.11. It explicitly shows the influence of the complete $\mathbf{X}$ on every hidden variable y_n and the sequential dependence of the individual y_n.

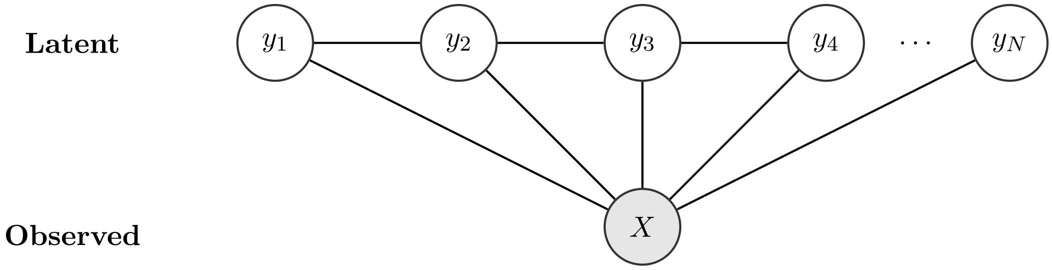


Figure 3.11: A schematic view of a linear chain CRF and its independencies.

A common application of CRFs is, similar to HMMs, part of speech tagging. Let us use this example to give a more concrete sense of how we might choose the feature function $\mathbf{f}$, summarized from Jurafsky and Martin (2022). Again, the value of $\mathbf{f}$ at position i can make use of all the information in $\mathbf{X}, y_n, y_{n-1}$. Therefore, some suitable components of the function (dimensions of the resulting vector) might be $\mathbb{1}\{x_n = the, y_n = DET\}$, where $\mathbb{1}\{\cdot\}$ is the indicator function. Such a feature might receive a strong weight $\mathbf{w}_i$ if the word "the" is independently very likely to be a determiner. Another example might be $\mathbb{1}\{y_n = VERB, y_{n-1} = AUX\}$, which would receive a large weight $\mathbf{w}_j$ if a verb is likely to follow an auxiliary verb. The dependence can of course go no longer than 1 step back, as in this case.

These were just two simple examples of components of $\mathbf{f}$. In practice such features are filled in automatically with feature templates, which result in millions of features like the ones above. With this, all we have to do is learn the weights $\mathbf{w}$ and then we can use the model to annotate sentences with POS tags (by searching for the most probable tags $\mathbf{Y}^*$).

Let us now also consider how to represent a (trained) CRF with a WFSA over the real semiring, as we did for N-grams and HMMs. As we will be working with arbitrary positive scores, we will not use the probability semiring but rather the real one. The idea is very similar to what we did before – the states of the WFSA represent the entire hidden state space and the transitions between states y and y' are weighted with the weights that the CRF scoring function assigns the hidden state y'

following the hidden state y, given that the observable input sequence is $\mathbf{X}$. You can imagine that every possible observable sequence $\mathbf{X}$ defines a new set of weights for the entire WFSA through the scoring function.

Combining everything, we can formally define the WFSA corresponding to a CRF as:

- $\Sigma = \{\varepsilon\}$
- $Q = \mathcal{Y}$
- $I = Q$
- $F = Q$
- $\delta = \{(y_n, \varepsilon, \exp\left[\mathbf{f}\left(y_n, y_j, \mathbf{X}\right)\right], y_j) \;\; \textit{for } y_n, y_j \in \mathcal{Y}\}$
- $\lambda : y_n \mapsto \exp\left[\mathbf{f}\left(y_n, \mathsf{BOS}, \mathbf{X}\right)\right] \; \forall y_n \in \mathcal{Y}$
- $\rho : y_n \mapsto \exp\left[\mathbf{f}\left(\mathsf{EOS}, y_n, \mathbf{X}\right)\right] \; \forall y_n \in \mathcal{Y}$

CRFs might remind you of the very well known logistic regression. Indeed, they are very similar. You can look at CFRs as a generalization of logistic regression to the prediction of sequences*, whereas the original logistic regression only models the probability of a* single *label.*

This concludes our quick tour of a few applications of WFSA to some well known problem. We can now look at the first binary operation and then, the intersection.

3.5 Intersection of Weighted Finite-state Automata

The material covered thus far allows us to introduce an efficient algorithm to compute the **intersection** of two WFSA. The algorithm will further serve as a constructive proof that weighted finite-state automata (and therefore weighted regular languages) are closed under intersection. Intersection is an important mechanism to construct intricate automata from more basic building blocks. Going from the intuition of an intersection of sets, the language of the intersection of the WFSA $\mathcal{A}_1$ and $\mathcal{A}_2$, denoted as $\mathcal{A}_1 \cap \mathcal{A}_2$, is the set of strings which is accepted by both of them, i.xerc., $L\left(\mathcal{A}_1 \cap \mathcal{A}_2\right) = L\left(\mathcal{A}_1\right) \cap L\left(\mathcal{A}_2\right)$. The *string weight* of a string $\boldsymbol{y}$ in the language of the intersection is the product of the string weights in each of the two input automata, i.e., $\left(\mathcal{A}_1 \cap \mathcal{A}_2\right)(\boldsymbol{y}) = \mathcal{A}_1(\boldsymbol{y}) \otimes \mathcal{A}_2(\boldsymbol{y})$.

The weighted intersection algorithm takes in two WFSA over the same semiring $\mathcal{W}$ with matching alphabets: $\mathcal{A}_1 = (\Sigma, Q_1, I_1, F_1, \delta_1, \lambda_1, \rho_1)$ and $\mathcal{A}_2 = (\Sigma, Q_2, I_2, F_2, \delta_2, \lambda_2, \rho_2)$ and outputs a WFSA $\mathcal{A} = (\Sigma, Q, I, F, \delta, \lambda, \rho)$[9]. Importantly, we will assume in this section that *the semiring $\mathcal{W}$ is commutative.*

Since the same string (or any suffix of it) can start from any state of the two automata independently, the algorithm has to consider all pairs of states of the two WFSA and the strings starting (or extending) from them. Therefore, the states in the intersection correspond to all pairs of states in the original WFSA. To ensure that a string $\boldsymbol{y}$ is only accepted in the intersection if it is accepted by both input WFSA, a transition with label a is added between two state pairs $(q_{11} \in Q_1, q_{12} \in Q_2)$ and $(q_{21} \in Q_1, q_{22} \in Q_2)$ if there is a transition with label a from q_{11} to q_{21} in $\mathcal{A}_1$ and a transition with label a from q_{12} to q_{22} in $\mathcal{A}_2$. Its weight in the intersection is the $\otimes$ multiplication of these transition weights in the original automata. That way, the score of a string $\boldsymbol{y}$ in the intersection $\mathcal{A}_1 \cap \mathcal{A}_2$ can be expressed as $\mathcal{A}(\boldsymbol{y}) = \mathcal{A}_1(\boldsymbol{y}) \otimes \mathcal{A}_2(\boldsymbol{y})$.

An example of the intersection of two WFSA, taken from Mohri (2009), is shown in Fig. 3.12.

We present two versions of the algorithm for computing the intersection. A *naïve version* which from the start considers all possible pairs of states and an *on-the-fly version*[10] which dynamically generates only accessible states of the intersection and does not consider non-accessible states. The basic version of the algorithm is relatively straight-forward. However, it *does not work* on automata with ε-transitions. An extension in the form of an additional component is needed to make it work in general and is deferred to the end of the section.

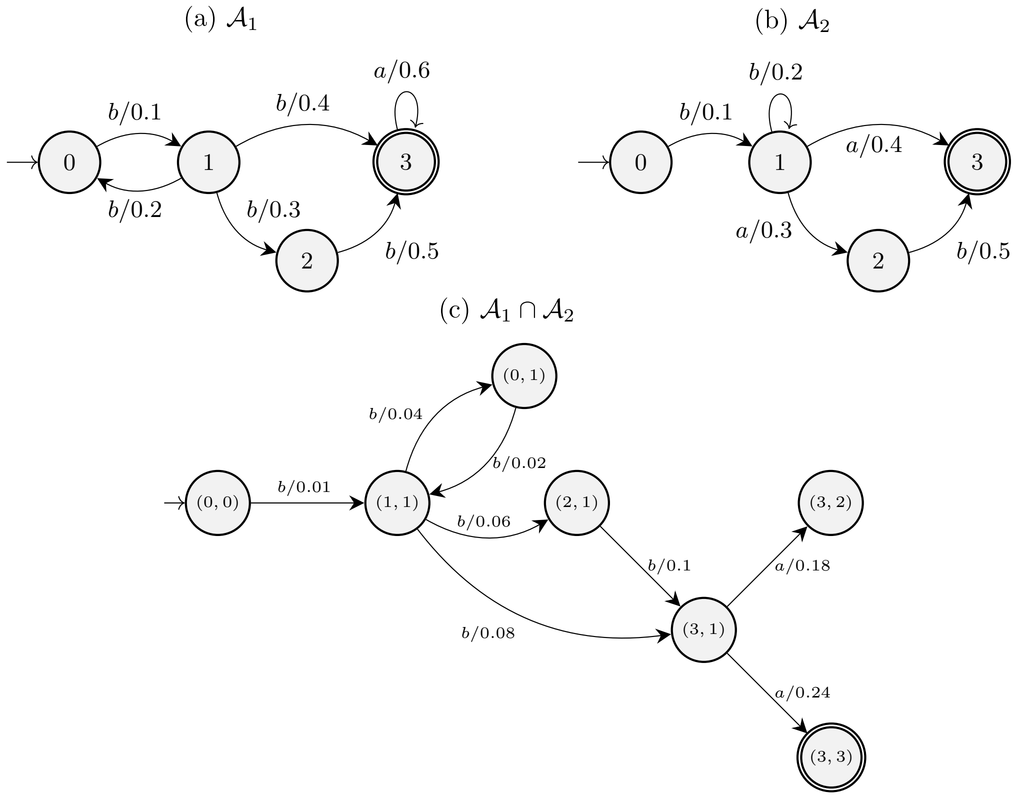
Naïve Algorithm

The naïve implementation is presented in Alg. 1. It loops through all possible pairs of states of the two input WFSA (Line 3) and considers all possible transitions with the same symbol leading from both states (Lines 4 and 5). In Lines 6 and 7, the appropriate edges are added to the intersection. Lines 9 and 10 simply initialize the starting and finishing states of the output WFSA, again weighted by the product of the starting/finishing weights of the states in the input automata.

It's easy to see this algorithm runs in quadratic time in the number of states and transitions: $\mathcal{O}\left(|\mathcal{A}_1||\mathcal{A}_2|\right)$.

[9] In a more general case, the input automata could have different alphabets. In that case, we restrict their alphabets to the intersection $\Sigma_1 \cap \Sigma_2$.

[10] Due to the way it construct the states of the intersection, we will also refer to this version as the "Accessible Intersection".


(a) $\mathcal{A}_1$

(b) $\mathcal{A}_2$

(c) $\mathcal{A}_1 \cap \mathcal{A}_2$

Figure 3.12: An example of the intersection of two WFSA. The WFSA shown in Fig. 3.12c is the untrimmed intersection of the WFSA from Fig. 3.12a and Fig. 3.12b. The initial and final weights are assumed to be **1** and not shown in the figures.

On-the-fly Algorithm or Accessible Intersection

The naïve version of the algorithm generates and loops through all possible pairs of states, some of which would never have been accessed in the intersection automaton and are thus useless. Thus, a natural speed-up is to combine the depth-first search used in accessibility computation with the intersection algorithm. Thus, we only add new states to the intersection machine when there is a path from an initial state (in the new machine) to that state. The algorithm starts by constructing all possible pairs of initial states and expands outwards searching for new accessible states. Since we start from any possible initial state pair, the algorithm considers all possible accessible states in $\mathcal{A}_1 \cap \mathcal{A}_2$ (and no others since we only look at the ones that can be reached). Note that, however, the produced states might still be non-*co*-accessible, since the algorithm can be lead into "dead-ends". Therefore, the resulting automaton is still not necessarily trim.

The implementation is very similar to the one in the naïve version and is presented in Alg. 2. We keep two additional data structures: a stack to keep all remaining state pairs which *can be reached* in the intersection but have not been processed yet and a set of all state pairs *already considered* to prevent going into the same state pair multiple times in the case of cycles. At each step of the while loop (Line 6), we consider the next state pair on the stack (which was found at some previous point in the traversal and is therefore reachable) instead of generating a new state pair from all possible ones. Other than that, the algorithm looks very much like the naïve version.

Depending on the number of states which can be accessed, the on-the-fly implementation might save the traversal of large parts of the resulting WFSA. However, the worst-case runtime is still

Algorithm 1 The naïve version of the algorithm for computing the intersection of two WFSA.

```
def NaïveIntersect($\mathcal{A}_1, \mathcal{A}_2$):
  $\mathcal{A} \leftarrow (\Sigma, Q, \delta, \lambda, \rho)$                ▷Create a new WFSA
  for $q_1, q_2 \in Q_1 \times Q_2$ :
    for $q_1 \xrightarrow{a_1/w_1} q_1', q_2 \xrightarrow{a_2/w_2} q_2' \in \mathcal{E}_{\mathcal{A}_1}(q_1) \times \mathcal{E}_{\mathcal{A}_2}(q_2)$ :
      if $a_1 = a_2$ :
        add states $(q_1, q_2)$ and $(q_1', q_2')$ to $\mathcal{A}$ if they have not been added yet
        add arc $(q_1, q_2) \xrightarrow{a_1/w_1 \otimes w_2} (q_1', q_2')$ to $\mathcal{A}$
  for $(q_1, q_2) \in Q$ :
    $\lambda_{\mathcal{A}} \leftarrow \lambda_1(q_1) \otimes \lambda_2(q_2)$
    $\rho_{\mathcal{A}} \leftarrow \rho_1(q_1) \otimes \rho_2(q_2)$
  return $\mathcal{A}$
```

Algorithm 2 On-the-fly (accessible) WFSA intersection.

```
def Intersect($\mathcal{A}_1, \mathcal{A}_2$):
  stack $\leftarrow [(q_1, q_2) \mid q_1 \in I_1, q_2 \in I_2]$
  visited $\leftarrow \{(q_1, q_2) \mid q_1 \in I_1, q_2 \in I_2\}$
  $\mathcal{A} \leftarrow (\Sigma, Q, \delta, \lambda, \rho)$                ▷Create a new WFSA
  while |stack| $> 0$ :
    $q_1, q_2 \leftarrow$ stack.pop()
    for $q_1 \xrightarrow{a_1/w_1} q_1', q_2 \xrightarrow{a_2/w_2} q_2' \in \mathcal{E}_{\mathcal{A}_1}(q_1) \times \mathcal{E}_{\mathcal{A}_2}(q_2)$ :
      if $a_1 = a_2$ :
        add states $(q_1, q_2)$ and $(q_1', q_2')$ to $\mathcal{A}$ if they have not been added yet
        add arc $(q_1, q_2) \xrightarrow{a_1/w_1 \otimes w_2} (q_1', q_2')$ to $\mathcal{A}$
        if $(q_1', q_2')$ not in visited :
          push $(q_1', q_2')$ to stack and visited
  for $(q_1, q_2) \in Q$ :
    $\lambda_{\mathcal{A}} \leftarrow \lambda_1(q_1) \otimes \lambda_2(q_2)$
    $\rho_{\mathcal{A}} \leftarrow \rho_1(q_1) \otimes \rho_2(q_2)$
  return $\mathcal{A}$
```

quadratic in the number of states and transitions: $\mathcal{O}\left(|\mathcal{A}_1||\mathcal{A}_2|\right)$.

We now prove the correctness of the on-the-fly version of the algorithm. We start by showing the following lemma, which is actually the bulk of the proof.

Lemma 3.5.1. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two ε-free WFSA over the same commutative semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$. Let $\mathcal{A} = \textsf{Intersect}\left(\mathcal{A}_1, \mathcal{A}_2\right)$. Then $\forall \boldsymbol{y} \in \Sigma^*, (q_1, q_2) \in Q : \ \boldsymbol{\alpha}_{\mathcal{A}}\left(\boldsymbol{y}, (q_1, q_2)\right) = \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y}, q_1\right) \otimes \boldsymbol{\alpha}_{\mathcal{A}_2}\left(\boldsymbol{y}, q_2\right)$.*

Proof. This is the first formal correctness proof of the course. As you will see, it sets up a common theme of the proofs for the first part of the course. Namely, we will often prove the correctness by inducing on the length of input string.

Base case $\boldsymbol{y} = a \in \Sigma \ (|\boldsymbol{y}| = 1)$

$$
\begin{aligned}
\boldsymbol{\alpha}_{\mathcal{A}}\left(\boldsymbol{y}, (q_1, q_2)\right) &= \boldsymbol{\alpha}_{\mathcal{A}}\left(a, (q_1, q_2)\right) \\
&= \bigoplus_{q_{I1} \in I_1, q_{I2} \in I_2} \ \bigoplus_{(q_{I1}, q_{I2}) \xrightarrow{a/w} (q_1, q_2)} \lambda\left((q_{I1}, q_{I2})\right) \otimes w && \text{(definition)} \\
&= \bigoplus_{q_{I1} \in I_1, q_{I2} \in I_2} \ \bigoplus_{(q_{I1}, q_{I2}) \xrightarrow{a/w} (q_1, q_2)} \lambda_1\left(q_{I1}\right) \otimes \lambda_2\left(q_{I2}\right) \otimes w && \text{(Line 14)} \\
&= \bigoplus_{q_{I1} \in I_1, q_{I2} \in I_2} \ \bigoplus_{q_{I1} \xrightarrow{a/w_1} q_1, q_{I2} \xrightarrow{a/w_2} q_2} \lambda_1\left(q_{I1}\right) \otimes \lambda_2\left(q_{I2}\right) \otimes w_1 \otimes w_2 && \text{(Line 10)} \\
&= \bigoplus_{q_{I1} \in I_1, q_{I2} \in I_2} \ \bigoplus_{q_{I1} \xrightarrow{a/w_1} q_1, q_{I2} \xrightarrow{a/w_2} q_2} \lambda_1\left(q_{I1}\right) \otimes w_1 \otimes \lambda_2\left(q_{I2}\right) \otimes w_2 && \text{(commutativity)} \\
&= \left(\bigoplus_{q_{I1} \in I_1} \ \bigoplus_{q_{I1} \xrightarrow{a/w_1} q_1} \lambda_1\left(q_{I1}\right) \otimes w_1 \right) \\
&\qquad \otimes \left(\bigoplus_{q_{I2} \in I_2} \ \bigoplus_{q_{I2} \xrightarrow{a/w_2} q_2} \lambda_2\left(q_{I2}\right) \otimes w_2 \right) && \text{(distributivity)} \\
&= \boldsymbol{\alpha}_{\mathcal{A}_1}\left(a, q_1\right) \otimes \boldsymbol{\alpha}_{\mathcal{A}_2}\left(a, q_2\right) && \text{(definition)}
\end{aligned}
$$

Inductive step Suppose now that $\boldsymbol{\alpha}_{\mathcal{A}}\left(\boldsymbol{y}\left(q_1, q_2\right)\right) = \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y}, q_1\right) \otimes \boldsymbol{\alpha}_{\mathcal{A}_2}\left(\boldsymbol{y}, q_2\right)$ for any $(q_1, q_2) \in Q$ and any string $\boldsymbol{y}$ such that $|\boldsymbol{y}| < n$. Let $\boldsymbol{y}$ be such that $|\boldsymbol{y}| = n - 1$ and $b \in \Sigma$. Then

$$
\begin{aligned}
\boldsymbol{\alpha}_{\mathcal{A}}\left(\boldsymbol{y} \circ b, \left(q_1', q_2'\right)\right) &= \bigoplus_{(q_1,q_2)\in Q} \boldsymbol{\alpha}_{\mathcal{A}}\left(\boldsymbol{y}, q\right) \otimes \bigoplus_{(q_1,q_2) \xrightarrow{b/w} \left(q_1',q_2'\right) \in \mathcal{E}((q_1,q_2))} w && \text{(algebraic manipulation)} \\
&= \bigoplus_{(q_1,q_2)\in Q} \boldsymbol{\alpha}_{\mathcal{A}}\left(\boldsymbol{y}, (q_1, q_2)\right) \otimes \bigoplus_{\substack{q_1 \xrightarrow{b/w_1} q_1' \in \mathcal{E}(q_1) \\ q_2 \xrightarrow{b/w_2} q_2' \in \mathcal{E}(q_2)}} w_1 \otimes w_2 && \text{(Line 9)} \\
&= \bigoplus_{(q_1,q_2)\in Q} \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y}, q_1\right) \otimes \boldsymbol{\alpha}_{\mathcal{A}_2}\left(\boldsymbol{y}, q_2\right) \otimes \bigoplus_{\substack{q_1 \xrightarrow{b/w_1} q_1' \in \mathcal{E}(q_1) \\ q_2 \xrightarrow{b/w_2} q_2' \in \mathcal{E}(q_2)}} w_1 \otimes w_2 && \text{(inductive hypothesis)} \\
&= \bigoplus_{(q_1,q_2)\in Q} \bigoplus_{\substack{q_1 \xrightarrow{b/w_1} q_1' \in \mathcal{E}(q_1) \\ q_2 \xrightarrow{b/w_2} q_2' \in \mathcal{E}(q_2)}} \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y}, q_1\right) \otimes \boldsymbol{\alpha}_{\mathcal{A}_2}\left(\boldsymbol{y}, q_2\right) \otimes w_1 \otimes w_2 && \text{(distributivity)} \\
&= \bigoplus_{(q_1,q_2)\in Q} \bigoplus_{\substack{q_1 \xrightarrow{b/w_1} q_1' \in \mathcal{E}(q_1) \\ q_2 \xrightarrow{b/w_2} q_2' \in \mathcal{E}(q_2)}} \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y}, q_1\right) \otimes w_1 \otimes \boldsymbol{\alpha}_{\mathcal{A}_2}\left(\boldsymbol{y}, q_2\right) \otimes w_2 && \text{(commutativity)} \\
&= \bigoplus_{(q_1,q_2)\in Q} \left(\bigoplus_{q_1 \xrightarrow{b/w_1} q_1' \in \mathcal{E}(q_1)} \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y}, q_1\right) \otimes w_1 \right) \\
&\qquad \otimes \left(\bigoplus_{q_2 \xrightarrow{b/w_2} q_2' \in \mathcal{E}(q_2)} \boldsymbol{\alpha}_{\mathcal{A}_2}\left(\boldsymbol{y}, q_2\right) \otimes w_2 \right) && \text{(distributivity)} \\
&= \left(\bigoplus_{q_1\in Q} \bigoplus_{q_1 \xrightarrow{b/w_1} q_1' \in \mathcal{E}(q_1)} \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y}, q_1\right) \otimes w_1 \right) \\
&\qquad \otimes \left(\bigoplus_{q_2\in Q} \bigoplus_{q_2 \xrightarrow{b/w_2} q_2' \in \mathcal{E}(q_2)} \boldsymbol{\alpha}_{\mathcal{A}_2}\left(\boldsymbol{y}, q_2\right) \otimes w_2 \right) && \text{(distributivity)} \\
&= \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y} \circ b, q_1'\right) \otimes \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y} \circ b, q_2'\right) && \text{(algebraic manipulation)}
\end{aligned}
$$

The lines marked as "algebraic manipulation" might require some additional work for a completely thorough treatment. This is left as an exercise.

■

We can now state the main theorem:

Theorem 3.5.1. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two ε-free WFSA over the same commutative semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$. Let $\mathcal{A} = \textsf{Intersect}\left(\mathcal{A}_1, \mathcal{A}_2\right)$. Then $\forall \boldsymbol{y} \in \Sigma^* : \mathcal{A}\left(\boldsymbol{y}\right) = \mathcal{A}_1\left(\boldsymbol{y}\right) \otimes \mathcal{A}_2\left(\boldsymbol{y}\right)$.*

Proof.

$$
\begin{aligned}
\mathcal{A}(\boldsymbol{y}) &= \bigoplus_{q \in Q} \boldsymbol{\alpha}_{\mathcal{A}}\left(\boldsymbol{y}, q\right) \otimes \rho(q) && \text{(definition)} \\
&= \bigoplus_{(q_1, q_2) \in Q} \boldsymbol{\alpha}_{\mathcal{A}}\left(\boldsymbol{y}, (q_1, q_2)\right) \otimes \rho\left((q_1, q_2)\right) && \text{(definition)} \\
&= \bigoplus_{(q_1, q_2) \in Q} \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y}, q_1\right) \otimes \boldsymbol{\alpha}_{\mathcal{A}_2}\left(\boldsymbol{y}, q_2\right) \otimes \rho\left((q_1, q_2)\right) && \text{(Lemma 3.5.1)} \\
&= \bigoplus_{(q_1, q_2) \in Q} \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y}, q_1\right) \otimes \boldsymbol{\alpha}_{\mathcal{A}_2}\left(\boldsymbol{y}, q_2\right) \otimes \rho_1\left(q_1\right) \otimes \rho_2\left(q_2\right) && \text{(Line 13)} \\
&= \bigoplus_{(q_1, q_2) \in Q} \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y}, q_1\right) \otimes \rho_1\left(q_1\right) \otimes \boldsymbol{\alpha}_{\mathcal{A}_2}\left(\boldsymbol{y}, q_2\right) \otimes \rho_2\left(q_2\right) && \text{(commutativity)} \\
&= \left(\bigoplus_{q_1 \in Q_1} \boldsymbol{\alpha}_{\mathcal{A}_1}\left(\boldsymbol{y}, q_1\right) \otimes \rho_1\left(q_1\right) \right) \otimes \left(\bigoplus_{q_2 \in Q_2} \boldsymbol{\alpha}_{\mathcal{A}_2}\left(\boldsymbol{y}, q_2\right) \otimes \rho_2\left(q_2\right) \right) && \text{(distributivity)} \\
&= \mathcal{A}_1\left(\boldsymbol{y}\right) \otimes \mathcal{A}_2\left(\boldsymbol{y}\right) && \text{(definition)}
\end{aligned}
$$

■

Handling ε-transitions

We now consider the extension of the intersection algorithm to automata containing ε-transitions. This section is slightly more technical and tedious, since we have to take care of a number of details. We start by investigating where a problem with ε-transitions might arise in the weighted case. Recall that we define the intersection of the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $\mathcal{A}\left(\boldsymbol{y}\right) = \mathcal{A}_1\left(\boldsymbol{y}\right) \otimes \mathcal{A}_2\left(\boldsymbol{y}\right)$. In particular, to calculate the resulting weight of the string $\boldsymbol{y}$ correctly, each pair of paths in $\mathcal{A}_1$ and $\mathcal{A}_2$ with the label $\boldsymbol{y}$ must correspond to *exactly one* path with label $\boldsymbol{y}$ in $\mathcal{A}$. Otherwise, we would sum over the multiple paths in $\mathcal{A}$ when calculating $\mathcal{A}\left(\boldsymbol{y}\right)$, which would result in an incorrect over-counted result over non-idempotent semirings.

Now, how can such an over-counting situation occur? It turns out that, when intersecting automata with ε-transitions, *redundant paths* are generated for a pair of paths in the original automata. We will present this situation on the example of the two automata shown in Fig. 3.13a and Fig. 3.13b.

Firstly, notice that "consuming an ε by one of the input automata" means that none of them move a symbol forward in the input string. It is therefore necessary to enable the input automata to consume ε "independently". Equivalently, we enable the original automata to stay in the same state whenever the other one consumes an ε. To do so, we *augment* the original automata as follows. We will modify the original ε-transitions and add new transitions with special symbols ε_1 and ε_2, which explicitly indicate what action each of the automata is taking. The ε-transitions in $\mathcal{A}_1$ are replaced by ε_2-transitions, while the ε-transitions in $\mathcal{A}_2$ are replaced by ε_1-transitions. Additionally, we add self loops to all states in both automata—ε_1-self loops in $\mathcal{A}_1$ and ε_2-self loops to $\mathcal{A}_2$. Notice that the opposite symbol is used for the self loops to the one used for original ε-transitions.

The semantics of these symbols are this: consuming ε_1 by the intersection $\mathcal{A}$ means that $\mathcal{A}_2$ consumed an ε while $\mathcal{A}_1$ stayed in the same state. Likewise, consuming ε_2 by the intersection means that $\mathcal{A}_1$ consumed an ε while $\mathcal{A}_2$ stayed in the previous state. This makes it clear why we need the extra self loops - without them, the original automata could only "move forward" over

ε-transitions *together* and take the same number of ε-transitions, ignoring the possibility of one automaton "waiting" for the other one while that one consumes ε's.

The augmentations of automata $\mathcal{A}_1$ and $\mathcal{A}_2$, which we denote as $\widetilde{\mathcal{A}_1}$ and $\widetilde{\mathcal{A}_2}$, are shown in Fig. 3.13c and Fig. 3.13d.

We can now actually take the intersection of $\widetilde{\mathcal{A}_1}$ and $\widetilde{\mathcal{A}_2}$. This, however, gives us a new problem—it introduces the redundant paths which we mentioned before. The automaton in Fig. 3.13e shows all plausible paths in the intersection of the augmented automata. All paths from state $(0, 0)$ to state $(4, 3)$ are equivalent, and we must only choose one. This is where the **epsilon filter** comes in. It is a finite-state automaton which "keeps track", for each state $q \in \widetilde{\mathcal{A}_1} \cap \widetilde{\mathcal{A}_2}$, what types of transitions have led to it (i.e., what symbols the original automata consumed), and thus, which transitions are still allowed to be taken in q to prevent overcounting.

Specifically, the epsilon filter is a **finite-state transducer**, another model of computation which we introduce in more detail later in §3.8. For now, it suffices to know how it differs from the automata we considered so far at a high level. In contrast to single-tape finite-state acceptors, the formalism we have discussed so far, transducers consider *pairs* of symbols at a time. One of them is said to be **consumed** and one is **emitted**. In our case, the state pairs will correspond to input symbols consumed by $\mathcal{A}_1$ and $\mathcal{A}_2$, simultaneously. The first symbol in the pair $a : b$ corresponds to the symbol consumed by $\mathcal{A}_1$ and the second one to the one consumed by $\mathcal{A}_2$. For example, the transition with labeled with $\varepsilon_1 : \varepsilon_2$ is taken when $\mathcal{A}_1$ consumes ε_1 (i.e., stays in the same state) while $\mathcal{A}_2$ consumes ε_2 (i.e., it takes an original ε-transition). It is shown in Fig. 3.13f. Rather than rigorously looking into how the filter is used,[11] let us understand the augmented operation from the modified on-the-fly/accessible intersection presented in Alg. 3.

The main idea of the filter's logic is to allow for only one path in the intersection for a given pair of paths in the original automata. We can see how it enforces that if we inspect its transitions. Starting in state 0, it disallows a matching $(\varepsilon_2, \varepsilon_2)$ immediately after $(\varepsilon_1, \varepsilon_1)$ since this can be done with fewer transitions instead via $(\varepsilon_2, \varepsilon_1)$. By symmetry, it also disallows a consuming $(\varepsilon_1, \varepsilon_1)$ immediately after $(\varepsilon_2, \varepsilon_2)$. In the same way, consuming $(\varepsilon_1, \varepsilon_1)$ immediately followed by $(\varepsilon_2, \varepsilon_1)$ is not permitted and should instead be done via $(\varepsilon_2, \varepsilon_1)\,(\varepsilon_1, \varepsilon_1)$. Similarly, $(\varepsilon_2, \varepsilon_2)\,(\varepsilon_2, \varepsilon_1)$ is ruled out. Try to think of which transitions the last two cases correspond to.

In the algorithm, the state q_f (f for filter) keeps, for each state pair in the intersection, the summary of what actions have been performed previously, relevant for that state pair. q_f gets updated with $\mathcal{TF}$'s logic with the ε-FILTER function, which is just the interface to the transducer's transition function. Lines 6–7 construct the augmented automata $\widetilde{\mathcal{A}_1}$ and $\widetilde{\mathcal{A}_2}$ with the renamed ε-transitions and added self-loops. Transitions from the augmented automata are then read on Line 16 and, for the matching pairs of symbols, a new state is added to the intersection just like in the unmodified algorithm. The filter state of the new state pair is updated *depending on the symbol pair which was used to reach it* and *the previous filter state in that target state pair.* This is the part which ensures we only consider one path in the intersection by blocking all transitions which end up in the **blocking state** $\perp$ of $\mathcal{TF}$.

Since the transition function implemented by ε-FILTER requires constant computation time, the runtime complexity of the modified algorithms is identical to the original intersection algorithm.

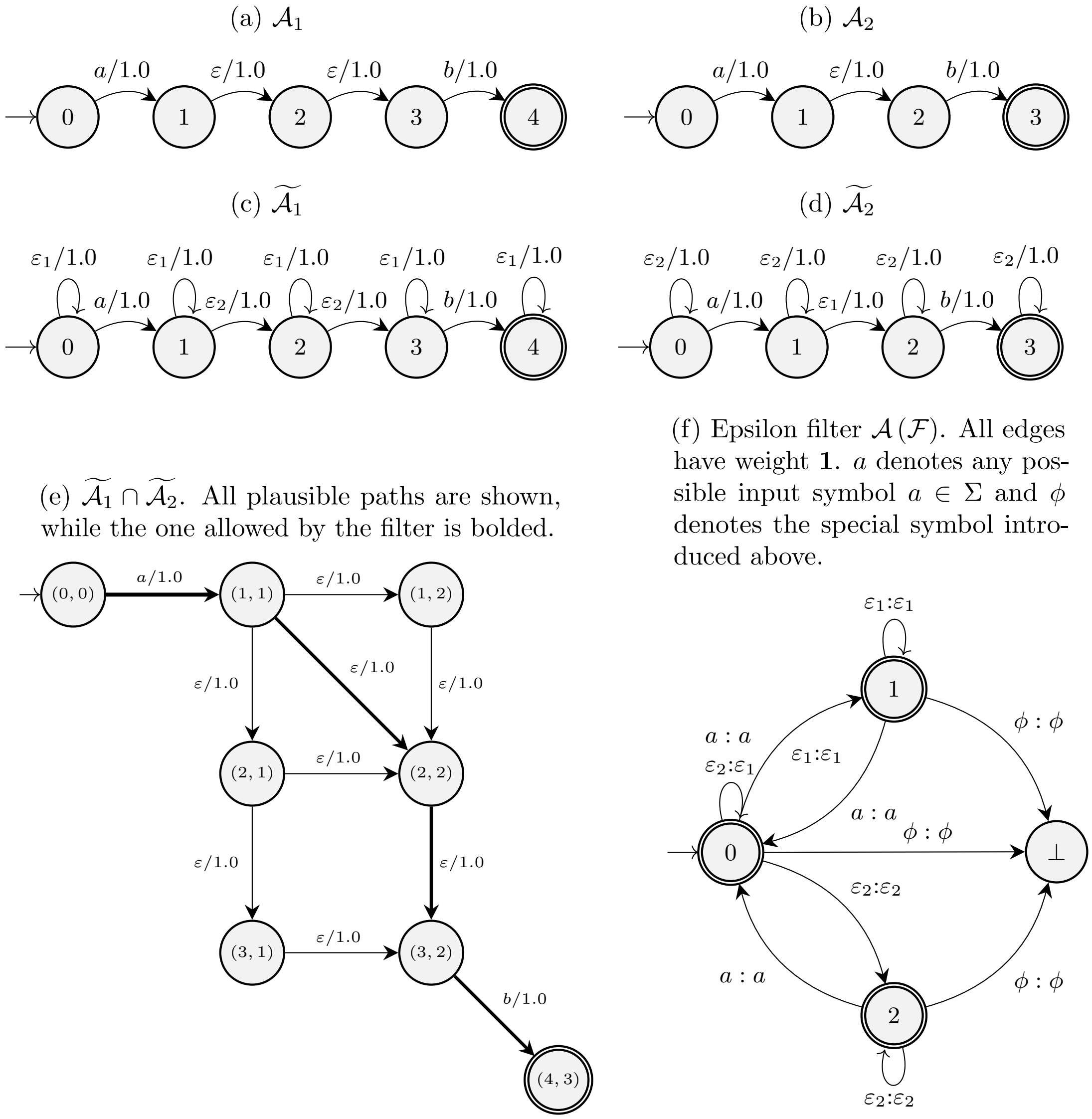
Notes on Proof of Correctness

Theorem 3.5.2. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two WFSA over the same alphabet Σ and the same commutative semiring $\mathcal{W}$. Let $\boldsymbol{\pi}_1$ be an arbitrary path in $\mathcal{A}_1$ and $\boldsymbol{\pi}_2$ be an arbitrary path in $\mathcal{A}_2$ with common*

[11] The operation we perform is actually a **weighted composition** of three weighted finite state transducers, $\mathcal{T}\widetilde{\mathcal{A}_1} \circ \mathcal{TF} \circ \mathcal{T}\widetilde{\mathcal{A}_2}$, where $\mathcal{T}\widetilde{\mathcal{A}}$ denotes the transformation of the automaton $\mathcal{A}$ into a transducer.

Algorithm 3 The on-the-fly version of the algorithm for computing the intersection of two WFSA with epsilon filter. The ε-FILTER function is the interface to the filter transducer $\mathcal{TF}$. ε-FILTER (a_1, a_2, q_f) returns the target state of the transition from q_f with the label $a_1 : a_2$ in $\mathcal{TF}$. Note that this is a version of the on-the-fly three-way composition of finite state transducers, which we will cover later (Allauzen and Mohri, 2008).

```
def $\epsilon$-filterIntersect($\mathcal{A}_1, \mathcal{A}_2$):
    $\mathcal{A} \leftarrow (\Sigma, Q, \delta, \lambda, \rho)$        ▷Initialize a new WFSA over the same semiring as $\mathcal{A}_1$ and $\mathcal{A}_2$.
    stack $\leftarrow \{(q_1, q_2, 0) \mid q_1 \in I_1, q_2 \in I_2\}$
    visited $\leftarrow \{(q_1, q_2, 0) \mid q_1 \in I_1, q_2 \in I_2\}$
    ▷ Rename $\varepsilon$-transitions in $\mathcal{A}_1$ and $\mathcal{A}_2$
    rename edges $q \xrightarrow{\varepsilon/w} q'$ to $q \xrightarrow{\varepsilon_2/w} q'$ in $\mathcal{A}_1$
    rename edges $q \xrightarrow{\varepsilon/w} q'$ to $q \xrightarrow{\varepsilon_1/w} q'$ in $\mathcal{A}_2$
    ▷ Add self loops
    for $q_1 \in Q_1$ :
        add edge $q_1 \xrightarrow{\varepsilon_1/\mathbf{1}} q_1$ to $\delta_1$
    for $q_2 \in Q_2$ :
        add edge $q_2 \xrightarrow{\varepsilon_2/\mathbf{1}} q_2$ to $\delta_2$
    ▷ We refer to the augmented input automata with (and their components) with $\widetilde{\mathcal{A}_1}$ and $\widetilde{\mathcal{A}_2}$
    ▷ Body of the algorithm
    while |stack| $> 0$ :
        $q_1, q_2, q_f \leftarrow$ stack.pop()
        for $q_1 \xrightarrow{a_1/w_1} q_1', q_2 \xrightarrow{a_2/w_2} q_2' \in \widetilde{\delta}_1 \times \widetilde{\delta}_2$ :
            if $\varepsilon$-FILTER $(a_1, a_2, q_f) \neq \perp$ :
                $q_f' \leftarrow \varepsilon$-FILTER $(a_1, a_2, q_f)$
                add arc $(q_1, q_2, q_f) \xrightarrow{a_1/w_1 \otimes w_2} (q_1', q_2', q_f')$ to $\mathcal{A}$
                if $\left(q_1', q_2', q_f'\right) \notin$ visited :
                    push $\left(q_1', q_2', q_f'\right)$ to stack
                    push $\left(q_1', q_2', q_f'\right)$ to visited
    ▷ Adding initial and final weights
    for $(q_1, q_2, q_f) \in Q$ :
        $\lambda(q_1, q_2, q_f) \leftarrow \widetilde{\lambda}_1(q_1) \otimes \widetilde{\lambda}_2(q_2)$
        $\rho(q_1, q_2, q_f) \leftarrow \widetilde{\rho}_1(q_1) \otimes \widetilde{\rho}_2(q_2)$
    return $\mathcal{A}$
```


(a) $\mathcal{A}_1$

(b) $\mathcal{A}_2$

(c) $\widetilde{\mathcal{A}_1}$

(d) $\widetilde{\mathcal{A}_2}$

(e) $\widetilde{\mathcal{A}_1} \cap \widetilde{\mathcal{A}_2}$. All plausible paths are shown, while the one allowed by the filter is bolded.

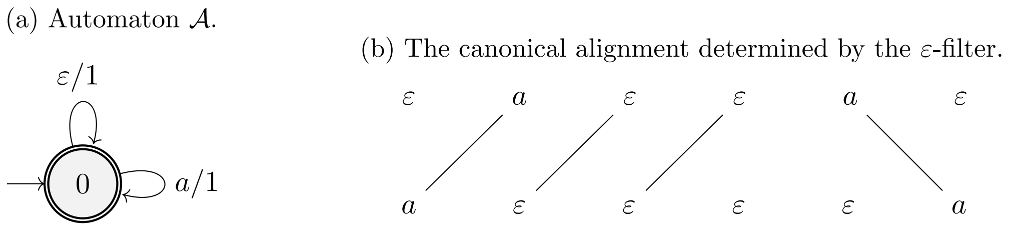
(f) Epsilon filter $\mathcal{A}(\mathcal{F})$. All edges have weight $\mathbf{1}$. a denotes any possible input symbol $a \in \Sigma$ and ϕ denotes the special symbol introduced above.

Figure 3.13: The automata $\mathcal{A}_1$ and $\mathcal{A}_2$ in Fig. 3.13a and Fig. 3.13b contain ε-transitions. Intersecting without taking into account path redundancy results in the automaton in Fig. 3.13e

yield $\boldsymbol{y} \in \Sigma^*$. Let $\mathcal{A}$ be the output of ϵ-filterIntersect *on $\mathcal{A}_1$ and $\mathcal{A}_2$. Then, there exists a unique path $\boldsymbol{\pi}$ in $\mathcal{A}$ that corresponds to the pair of paths $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$. Moreover, the weight of this path in $\mathcal{A}$ in $w(\boldsymbol{\pi}_1) \otimes w(\boldsymbol{\pi}_2)$.*

Proof. ("Proof" Sketch) As discussed in the main text, the only problem that can arise is over-counting due to ambiguity with respect to ε-transition. We get around this over-counting by canoncalizing the paths with the filter machine. We provide a simple example that shows how this works in practice. Consider the one-state automaton $\mathcal{A}$ shown in Fig. 3.14a. We will look at the intersection of $\mathcal{A}$ with itself. The accepting paths in $\mathcal{A}$, and thus $\mathcal{A} \cap \mathcal{A}$, are of the form $\{a, \varepsilon\}^*$.

For every pair of accepting paths *with the same yield*, we need *exactly one* accepting path in $\mathcal{A} \cap \mathcal{A}$. This is where the crucial part of the ε-filter comes in. It helps us achieve what we need by "canonicalizing" the alignment of the ε-transitions along the paths, i.e., ensuring that for a the paths with the same yield, only one path in the intersection will be chosen. Fig. 3.14b shows how the alignment would be chosen for a specific pair of paths. With some thinking, it is easy to see how such an alignment would be mapped to a path through the state space of $\mathcal{A} \cap \mathcal{A}$.


(a) Automaton $\mathcal{A}$.

(b) The canonical alignment determined by the ε-filter.

Figure 3.14: A schema of how ε-aware intersection must work.

A more rigorous argument can be achieved with additional work.

■

3.6 The Pathsum

The main difference between the treatment of automata you may have seen in previous courses and the treatment in this course is the fact that our focus is on *weighted* automata. Indeed, it is the mix of weights with automata theory that makes them naturally applicable to machine learning. In this lecture, we are going to focus on the development for efficient algorithms for the computation of **pathsums** in semiring-weighted WFSAs. This is a very general treatment of a broad class of problems which can be framed as a sum over path weights for some domain-specific notions of multiplication of weights along a path and the sum over the path weights—hence, the name *algebraic path problem*. In the tropical semiring, for example, this would correspond to a minimum over the additive path lengths.[12]

We start with the definition of the pathsum problem and then look at plethora of algorithms for computing it in different classes of WFSA. These problems turn out to be crucial in the design of several algorithms such as determinization or weight pushing and in many other contexts.

A Note on Notation. This chapter introduces algorithms used to compute quantities related to WFSA as introduced in §3.2. However, many of these are pure graph algorithms that work directly on a (directed) weighted graph, without the interpretation of the underlying WFSA it represents. Therefore, we will sometimes resort to simpler notation that ignores the WFSA. In concrete terms, this means that we will not make use of the labels on the edges.

A WFSA $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ has two natural notions of "size." First, is the number of states, which we denote $|Q|$, and the second is the number of transitions, which we denote $|\delta|$. However, in order to draw a tight connection with graph algorithms that you may already be familiar with, we will use V (for vertices) to refer to the number of states and E (for edges) to refer to the number of transitions whenever we are talking about pure graph algorithm.

3.6.1 The Pathsum

Let us start by introducing the general pathsum problem.

Definition 3.6.1 (Pathsum)**.** *Let* $\Pi(\mathcal{A})$ *denote the set of paths in an automaton* $\mathcal{A}$ *over a semiring* $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$*. The* ***pathsum*** *in* $\mathcal{A}$ *is the* sum over all these paths*, formally defined as*

$$Z(\mathcal{A}) \overset{\text{def}}{=} \bigoplus_{\boldsymbol{\pi} \in \Pi(\mathcal{A})} w(\boldsymbol{\pi}) = \bigoplus_{\boldsymbol{\pi} \in \Pi(\mathcal{A})} \lambda(p(\boldsymbol{\pi})) \otimes w_I(\boldsymbol{\pi}) \otimes \rho(n(\boldsymbol{\pi})) \tag{3.27}$$

Additionally, we define the pathsum between two states $q_1, q_2 \in Q$ *as*

$$Z(q_1, q_2) \overset{\text{def}}{=} \bigoplus_{\boldsymbol{\pi} \in \Pi(q_1, q_2)} w(\boldsymbol{\pi}) = \bigoplus_{\boldsymbol{\pi} \in \Pi(q_1, q_2)} \lambda(p(\boldsymbol{\pi})) \otimes w_I(\boldsymbol{\pi}) \otimes \rho(n(\boldsymbol{\pi})) \tag{3.28}$$

The notation $\mathrm{Z}(\mathcal{A})$ comes from statistical physics, where the normalizing constants are often denoted with Z for the German word *Zustandssumme*, meaning *configuration sum*. As we will see, the pathsum can often be used for computing normalizing constants of distributions over structured entities, hence the name.

In general, there may be an infinite number of accepting paths in a WFSA. Indeed, every cyclic WFSA has an infinite number of paths. Despite that, under very general conditions, the

[12] See, for example https://en.wikipedia.org/wiki/Shortest_path_problem#General_algebraic_framework_on_semirings:_the_algebraic_path_problem

pathsum can still be efficiently computed! We will also see how the pathsum problem subsumes computational problems found in machine learning, e.g., normalizing an n-gram language model, computing marginals in a hidden Markov model and normalizing a conditional random field.

Moreover, this quantity comes up in many other tasks and algorithms (with specific semiring instances and under different names), for example:

(i) Shortest path in a graph (tropical semiring).

(ii) Sum of all paths (real semiring).

(iii) Transforming a WFSA to a weighted regular expression (Kleene semiring).

3.6.2 Pathsums in Acyclic Machines

We will start our investigation by focusing on computing the pathsum for the special case of an *acyclic* finite-state machine over *any* semiring. The absence of directed cycles means that there is only a finite number of accepting paths, which makes the first set of algorithms we consider conceptually simple. We will also observe that their runtime is asymptotically faster than the runtime of the algorithms for the general cyclic case.

To be able to efficiently compute the pathsum of an acyclic graph, we need the notion of a topological sort.

Definition 3.6.2 (Topological Sort)**.** *A **topological sort** of a directed graph is a linear ordering of its vertices such that for every directed edge $q \rightarrow q'$, we have that q precedes q' (denoted with $q \preccurlyeq q'$) in the ordering, i.e., children follow their parents in the ordering. Recall an ordering is linear if every pair of elements is comparable, $\forall q, q' \in Q: \; q \preccurlyeq q'$ **or** $q' \preccurlyeq q$.*

Importantly, if we traverse the nodes in a topological order, we are guaranteed to have encountered all of a q' predecessors by the time we visit q. Clearly, a topological sort exists only for graphs which contain no directed cycles, i.e., directed acyclic graphs (DAGs). A topological sort is not necessarily unique.

Example 3.6.1. *A topological sort of the states of the WFSA in Fig. 3.15 is* $\langle 3, 1, 4, 6, 5, 2 \rangle$.

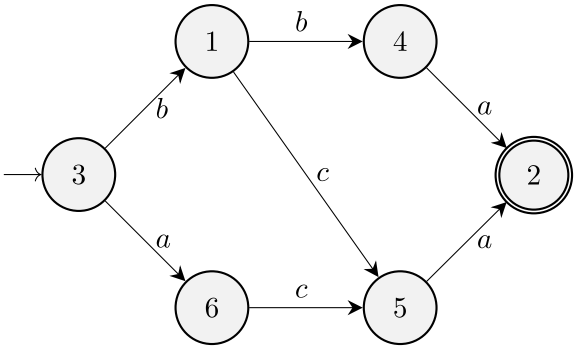


Figure 3.15: An acyclic boolean WFSA.

Computing a topological sort of a DAG is relatively straight-forward. It relies on a depth-first traversal of the graph from a starting node and sorts the nodes based on the quantities computed during the traversal. Before we describe topological sorting in more detail, we will quickly recall the classic depth-first search algorithm and with it the concept of node discovery and finishing times.

Depth-first search. Depth-first search (DFS) is an algorithm for traversing graphs which works by starting at an arbitrarily selected root node and explores as far as possible along each branch before backtracking (the complete exploration of all children of a a node q is called the **expansion** of q). The algorithm can be restarted at multiple nodes until the entire graph is traversed. When running a DFS on a WFSA, the only paths we are interested in are the ones starting at some node $q_I \in I$. Therefore, we only ever restart from there.

The order of traversal defines two important quantities of each node in the graph, its **discovery** and **finishing** times. The discovery time of a node q, denoted as $d\,[q]$, is the number of nodes discovered or finished before first seeing q and the finishing time of q, denoted as $f\,[q]$, is the number of nodes discovered or finished before finishing the expansion of q.

Algorithm 4 The DepthFirstSearch algorithm

```
def DepthFirstSearchRecursion($q$):
    add $q$ to in-progress
    for $q' \in \mathcal{E}(q)$ :
        if $q' \notin$ in-progress and $q' \notin$ finished :
            DepthFirstSearchRecursion($q'$)
    remove $q$ from in-progress
    add $q$ to finished
    finished[$q$] $\leftarrow$ counter
    counter $\leftarrow$ counter $+ 1$
def DepthFirstSearch($\mathcal{A}$):
    finished $\leftarrow \{\}$                    ▷Initialize an empty map
    in-progress $\leftarrow \{\}$                 ▷Initialize an empty set
    counter $\leftarrow 0$
    for $q_I \in I$ :
        if $q_I \notin$ in-progress :
            DepthFirstSearchRecursion($q_I$)
    return finished
```

For completeness, the version of DepthFirstSearch suitable for WFSA is outlined in Alg. 4.

After running DepthFirstSearch on a DAG, the finishing times give us all the information we need to sort the graph topologically. Indeed, all we need to do is to sort the nodes by decreasing finishing time and we get the topological sort of the graph. The steps are outlined in Alg. 5.

Algorithm 5 The TopologicalSort algorithm

```
def TopologicalSort($\mathcal{A}$):
    finished $\leftarrow$ DepthFirstSearch ($\mathcal{A}$)
    sort finished according to the finishing times in descending order
    return finished
```

The backward algorithm

A Linear-time Dynamic Program. We are now in a position to explain the basic dynamic program for computing the pathsum in an acyclic machine. The idea, in words, is to sort the vertices topologically and then proceed in reverse topological order by, $\forall q \in Q$, $\oplus$-summing up the weights of all the paths coming from the node q and ending in a final state $q_F \in F$. We make heavy use of the distributivity property of the semiring while doing so. Traditionally, the dynamic program for computing the pathsum in a WFSA is called the **backward algorithm**. The procedure owes its name to the fact that one traverses the arcs of a WFSA in *reverse* topological order, i.e., one traverses the arcs in a backwards fashion.

To illustrate the quantities computed by the algorithm, consider Fig. 3.16. When calculating $\boldsymbol{\beta}(q)$, we $\oplus$-sum over all the children's $\boldsymbol{\beta}$ values, weighting them with the weights of the edge connecting them to q. Since the $\boldsymbol{\beta}$ values of the children will have been computed already, this corresponds to summing over all outgoing paths from q, using the distributivity property, as we will see in the proof of the following theorem.

Theorem 3.6.1. *Let $\mathcal{A}$ be an acyclic WFSA weighted with semiring $\mathcal{W}$. Then, for the automaton*

Algorithm 6 The backward algorithm.

```
1. def backward($\mathcal{A}$, $\mathcal{W}$):
2.     for $q \in$ rtopo($\mathcal{A}$) :                                  ▷ We proceed in reverse topological order.
3.         $\boldsymbol{\beta}(q) = \rho(q) \oplus \bigoplus_{q \xrightarrow{a/w} q'} w \otimes \boldsymbol{\beta}(q')$      ▷ Since $q' \preccurlyeq q$, $\boldsymbol{\beta}(q')$ is already computed.
4.     return $\bigoplus_{q_I \in I} \lambda(q_I) \otimes \boldsymbol{\beta}(q_I)$
```

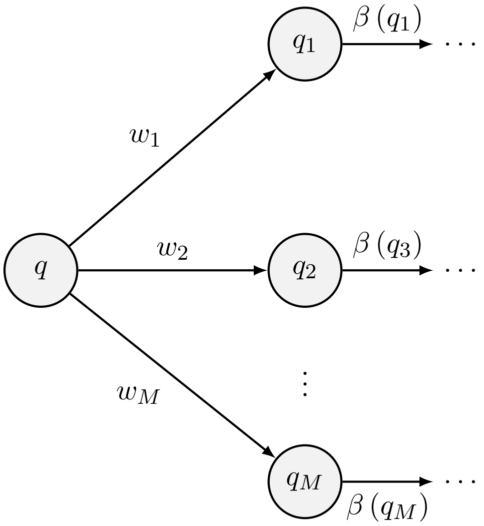


Figure 3.16: The general idea of the backward algorithm. The backward weight $\boldsymbol{\beta}(q)$ is computed *after* the backward weights $\boldsymbol{\beta}(q_1), \ldots, \boldsymbol{\beta}(q_m)$ as we proceed in reverse topological order. Above we have $\boldsymbol{\beta}(q) = \rho(q) \oplus \bigoplus_{m=1}^{M} w_m \otimes \boldsymbol{\beta}(q_m)$.

$\mathcal{A}$, backward correctly computes the backward weights ($\boldsymbol{\beta}$) and, thereby, the pathsum. Moreover, the algorithm runs in $\mathcal{O}(E)$ time and uses $\mathcal{O}(V)$ space.

Proof. First, we prove that Alg. 6 correctly computes the backward weights by induction on the number of iterations of the algorithm. Let $\langle q_1, \ldots, q_N \rangle$ be an explicit reverse topological ordering of the states the WFSA.

Base Case. Suppose the WFSA $\mathcal{A}$ has k vertices with no outgoing arcs. Since $\mathcal{A}$ is acyclic, we have $k \geq 1$. Call these states $\{q_1, \ldots, q_k\}$. By the definition of a topological sort, these k states must be the first k elements in the reverse topological sort. Thus, Line 3 of Alg. 6 correctly computes the backward weight for the first k iterations.

Inductive Step. We assume the correctness of the algorithm for its first $n - 1 \geq k$ steps, which processes the first $(n - 1)$ states in the reverse topological order. Below, we derive a recursion relation, that shows, if $\boldsymbol{\beta}(q_{n-1})$ is correctly computed, then $\boldsymbol{\beta}(q_n)$ must be as well. The derivation is as follows:

$$\begin{aligned}
\boldsymbol{\beta}(q_n) &= \rho(q_n) \oplus \bigoplus_{\boldsymbol{\pi} \in \Pi(q_n)} w_{\mathrm{I}}\left(\boldsymbol{\pi}\right) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) && (3.29)\\
&= \rho(q_n) \oplus \bigoplus_{q_n \xrightarrow{a/w} q_{n-1}} w \otimes \bigoplus_{\boldsymbol{\pi} \in \Pi(q_{n-1})} w_{\mathrm{I}}\left(\boldsymbol{\pi}\right) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) && (3.30)\\
&= \rho(q_n) \oplus \bigoplus_{q_n \xrightarrow{a/w} q_{n-1}} w \otimes \boldsymbol{\beta}(q_{n-1}) \qquad \text{(inductive hypothesis)} && (3.31)
\end{aligned}$$

Thus, $\boldsymbol{\beta}(q_n)$ is correct As we can see, Eq. (3.31) mirrors Line 3 of Alg. 6. Thus, we see that Alg. 6 correctly computes the backwards weights. Finally, to turn a correct computation of the backwards weights into a pathsum, we need to pre-multiply the backward weights with the initial weights

$$\bigoplus_{q \in Q} \lambda\left(q\right) \otimes \boldsymbol{\beta}(q) \tag{3.32}$$

which is computed on Line 5 of Alg. 6.

With respect to the runtime, it is apparent from the pseudocode that each edge is touched exactly one, which gives us a bound of $\mathcal{O}\left(E\right)$. ■

Note that there is nothing special in going *backward* through the automaton. We could have just as easily proceeded in the forward topological order, visited the *incoming* edges of the nodes and $\oplus$-added the *initial* weights instead of the final ones (which would be post-multiplied at the end). In fact, the same algorithm applied in the forward direction is very appropriately called the forward algorithm. It is also customary to denote the dynamic programming table with $\boldsymbol{\alpha}$ in that case instead of $\boldsymbol{\beta}$.

3.6.3 Closed Semirings

We now seek to generalize the above dynamic program to the cyclic case. This involves a bit more formal machinery than we have introduced so far. Namely, we need to develop the idea of a **closed semiring**.

Definition 3.6.3. *A **closed** semiring is a semiring which is* augmented with an additional unary operation*. The additional unary operation is called the **Kleene star** and is denoted with a superscript asterisk, e.g., x^*. We use the verb* to asterate *to describe the application of the Kleene star. The Kleene star must obey the following two axioms:*

(i) $x^* = \boldsymbol{1} \oplus x \otimes x^*$;

(ii) $x^* = \boldsymbol{1} \oplus x^* \otimes x$.

Note *There are multiple formulations of a closed semiring. Some sources require a closed semiring to satisfy another property, namely that for any countable index set $\mathcal{I}$ and set $\{x_i;\ i \in \mathcal{I}\} \subseteq \mathbb{K}$, the sum $\bigoplus_{i \in \mathcal{I}} x_i$* exists *and* is unique *- the order of the additions must be irrelevant. We follow the definition from Lehmann (1977) and do not require this condition. We could have also more generally defined closed semirings by adding the Kleene star operator to algebraic structures which are not even semirings themselves—sometimes the annihilation axiom is left out from the definition of a closed semiring. To keep things simple, we are keeping it. Moreover, closeness can also be defined with a very different set of axioms, see Mohri et al. (2002). Some sources also refer to our definition of closed semirings as **star semirings** (Droste et al., 2009) alluding to the operation *.*

The above definition is a bit abstract. However, it comes from an axiomatization of a very familiar concept: the geometric series. Extending pathsums to the case of cyclic WFSAs requires us, definitionally, to sum over an *infinite* number of paths. Recall that for any real value $x \in [0, 1)$, we have $\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$. We will later see that this is analogous to summing the weights of an infinite number of paths over a directed cycle. We will show that this definition satisfies the two axioms above.

Proposition 3.6.1.

$$x^* = \sum_{n=0}^{\infty} x^n = \frac{1}{1-x} \tag{3.33}$$

is a valid Kleene star.

Proof. We just have to show that the above axioms hold:

$$\begin{aligned} x^* = \frac{1}{1-x} &= 1 + \frac{1}{1-x} - 1 = 1 + \left(\frac{1}{1-x} - 1\right) \\ &= 1 + \frac{1-1+x}{1-x} = 1 + \frac{x}{1-x} = 1 + x\frac{1}{1-x} = 1 + xx^* \end{aligned}$$

Since the multiplication operation on the reals is commutative, this directly implies the second axiom as well, meaning that both hold. Thus, we have shown that $x^* = \sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$ is a valid Kleene star. ■

The Kleene star operator can be *constant* in some semirings.

Example 3.6.2. *The boolean semiring with $x^* = 1$.*

Example 3.6.3. *The tropical semiring with $x^* = 0$.*

We next introduce a few more classes of semirings which we will make use of shortly. The first ones are **idempotent** semirings.

Definition 3.6.4 (Idempotent Semiring)**.** *A semiring is* ***idempotent*** *if, for every element $x \in A$, we have $x \oplus x = x$.*

Definition 3.6.5 (k-closed Semiring)**.** *For $k \in \mathbb{Z}_{\geq 0}$, a semiring $(\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ is k**-closed** if*

$$\forall x \in \mathbb{K}, \bigoplus_{n=0}^{k+1} x^{\otimes n} = \bigoplus_{n=0}^{k} x^{\otimes n}. \tag{3.34}$$

Here we define

$$x^{\otimes n} \overset{\text{def}}{=} \underbrace{x \otimes x \otimes \cdots \otimes x}_{n \text{ times}} \tag{3.35}$$

Intuitively, this means that adding powers of elements $> k$ does not affect the sum anymore. This makes any infinite geometric sum in fact finite, as the following lemma shows.

Lemma 3.6.1. *Let $\mathcal{W}$ be a k-closed semiring. Then $\forall l > k$:*

$$\forall x \in \mathbb{K}, \bigoplus_{n=0}^{l} x^{\otimes n} = \bigoplus_{n=0}^{k} x^{\otimes n}. \tag{3.36}$$

Proof. We prove the lemma with induction on l. By definition, the lemma holds for $l = k + 1$. Now suppose the lemma holds for some l. Then

$$\begin{aligned}
\bigoplus_{n=0}^{l+1} x^{\otimes n} &= \bigoplus_{n=0}^{l-k-1} x^{\otimes n} \oplus \bigoplus_{n=l-k}^{l+1} x^{\otimes n} \\
&= \bigoplus_{n=0}^{l-k-1} x^{\otimes n} \oplus x^{\otimes l-k} \otimes \bigoplus_{n=0}^{k+1} x^{\otimes n} \\
&= \bigoplus_{n=0}^{l-k-1} x^{\otimes n} \oplus x^{\otimes l-k} \otimes \bigoplus_{n=0}^{k} x^{\otimes n} \\
&= \bigoplus_{n=0}^{l-k-1} x^{\otimes n} \oplus \bigoplus_{n=l-k}^{l} x^{\otimes n} \\
&= \bigoplus_{n=0}^{l} x^{\otimes n} = \bigoplus_{n=0}^{k} x^{\otimes n}
\end{aligned}$$

where the last line holds by the inductive hypothesis. ■

The next lemma shows that k-closed semirings are in fact a broader class of semirings which subsumes the closed semirings introduced above.

Lemma 3.6.2. *For any k, a k-closed semiring is closed.*

Proof. Let $\mathcal{W}$ be a k-closed semiring. We define the Kleene star operator x^* as

$$x^* = \bigoplus_{n=0}^{k} x^{\otimes n}, \ \forall x \in \mathbb{K}. \tag{3.37}$$

We can verify that the operation satisfies the two axioms:

$$x^* = \bigoplus_{n=0}^{k} x^{\otimes n} = \bigoplus_{n=0}^{k+1} x^{\otimes n} = x^0 + \bigoplus_{n=1}^{k+1} x^{\otimes n} = \mathbf{1} + x \otimes \bigoplus_{n=0}^{k} x^{\otimes n} = \mathbf{1} + x \otimes x^*$$

The proof of the second axiom is symmmetric. The 4^{th} step in the equation uses the fact that $\mathcal{W}$ is a k-closed, while the other ones are just basic operations in the semiring. ■

The proof is also constructive in the sense that it gives us an explicit construction of the Kleene star operator for any k-closed semiring, namely

$$x^* = \bigoplus_{n=0}^{k} x^{\otimes n}, \ \forall x \in \mathbb{K}. \tag{3.38}$$

Example 3.6.4. *Concretely, this means, $\forall x \in \mathbb{K}$:*

- *If semiring is* 0*-closed:*

$$x^* = \mathbf{1}$$

- *If semiring* 1*-closed:*

$$x^* = \mathbf{1} \oplus x$$

- *If semiring* 2*-closed:*

$$x^* = \mathbf{1} \oplus x \oplus x^{\otimes 2}$$

An important special case of k-closed semirings are in fact 0-closed semirings, in which, by definition, $\forall x \in \mathbb{K}, x^* = \mathbf{1}$. They are also referred to as **simple semirings**. Simple semirings are special cases of idempotent semirings, as the following proposition shows.

Proposition 3.6.2. *A* 0*-closed semiring is idempotent.*

Proof. In a 0-closed semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, by definition $\mathbf{1} \oplus x = \mathbf{1} \ \forall x \in \mathbb{K}$. In particular $\mathbf{1} \oplus \mathbf{1} = \mathbf{1}$. (Left) multiplication with $x \in \mathbb{K}$ then gives us $x \otimes (\mathbf{1} \oplus \mathbf{1}) = x \otimes \mathbf{1} \implies x \oplus x = x$. ■

The converse does *not* hold, i.e., there are idempotent semirings which are not 0-closed, as the next example shows.

Example 3.6.5. *The tuple $\mathcal{W} = (\mathbb{R}_+, \max, \cdot, 0, 1)$ defines a semiring. Since $\max(x, x) = x, \forall x \in \mathbb{R}_+$, the semiring is idempotent. However, $1 \oplus x = \max(1, x) = x \neq 1$ in general, therefore $\mathcal{W}$ is not* 0*-closed.*

The boolean semiring and the tropical semiring are examples of 0-closed semirings (Mohri, 2009). Simple semirings have an especially attractive and understandable intuition in the tropical semiring $(\mathbb{R}_+, \min, +, \infty, 0)$ where they capture the idea that going through cycles in a graph only increases the path length. Indeed, since by definition the weight of a path of length zero is 0 (which is the semiring $\mathbf{1}$ in the tropical semiring) and $\mathbf{1} \oplus x = \mathbf{1}$, i.e., $\min(0, x) = 0$, taking a cycle will always increase the path length.

3.6.4 The General Pathsum Problem

In this section, we will show that, should our WFSA be defined over a closed semiring, we can develop an efficient algorithm to sum over the paths in cyclic machines. Our exposition will largely follow Gondran and Minoux (2008), but the ideas are generally attributed to Lehmann (1977) and Tarjan (1981), who both introduced an elegant generalization of the Floyd–Warshall algorithm for the all-pairs shortest path problem. Indeed, we will see how Floyd–Warshall is just an instance of Lehmann's algorithm with a specific closed semiring: the Tropical semiring.

Semirings over Matrices

This section demonstrates how we can lift a semiring structure over a set to a semiring structure over matrices with elements from that set. We give definitions of matrix addition and multiplication in the semiring and then show that these form a semiring.

Definition 3.6.6 (Semiring-based matrix addition)**.** *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be* $D \times D$ *matrices with entries in semiring* $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$*. Then, the sum of the matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *is defined as*

$$(\mathbf{A} \oplus \mathbf{B})_{ij} \overset{\text{def}}{=} \mathbf{A}_{ij} \oplus \mathbf{B}_{ij} \quad i, j \in [D] \tag{3.39}$$

Definition 3.6.7 (Semiring-based matrix multiplication)**.** *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be* $D \times D$ *matrices with entries in semiring* $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$*. Then, the product of the matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *is defined as*

$$(\mathbf{A} \otimes \mathbf{B})_{ij} \overset{\text{def}}{=} \bigoplus_{k=1}^{D} \mathbf{A}_{ik} \otimes \mathbf{B}_{kj} \quad i, j \in [D] \tag{3.40}$$

As you can see, the semiring-based matrix multiplication is a straight-forward generalization of the normal matrix multiplication in the field of reals whereas the semiring-based matrix addition is simply defined element-wise.

We will see that any semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ can be "lifted" into a semiring over the matrices. In fact, the following straight-forward proposition tells us exactly that.

Proposition 3.6.3. *Let* $\mathbf{M}_D(\mathcal{W})$ *be the set of* $D \times D$ *matrices* $\mathbf{M}$ *with entries* $\mathbf{M}_{ij} \in \mathbb{K}$*. Then* $(\mathbf{M}_D(\mathcal{W}), +, \cdot, \mathbf{O}, \mathbf{I})$*, where* $\mathbf{O}$ *is a matrix of all zeros and* $\mathbf{I}$ *is the* $D \times D$ *identity matrix with the diagonal entries being the semiring* ***1*** *and the off-diagonals being the semiring* ***0****, is also a semiring.*

Proof. The proof is straight-forward and involves checking that the the two operations defined in Eq. (3.39) and Eq. (3.40) together with the specified units satisfy the semiring axioms. It is fact no different than the proof that real matrices form a ring (without the additive inverses). We just show one of the axioms, namely, that the matrix multiplication is associative, and leave the rest as an exercise.

$$
\begin{aligned}
\mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}) &= \bigoplus_{k=1}^{D} \mathbf{A}_{ik} \otimes (\mathbf{B} \otimes \mathbf{C})_{kj} && \text{(definition)} \\
&= \bigoplus_{k=1}^{D} \mathbf{A}_{ik} \otimes \left(\bigoplus_{l=1}^{D} \mathbf{B}_{kl} \otimes \mathbf{C}_{lj} \right) && \text{(definition)} \\
&= \bigoplus_{k=1}^{D} \bigoplus_{l=1}^{D} \mathbf{A}_{ik} \otimes (\mathbf{B}_{kl} \otimes \mathbf{C}_{lj}) && \text{(distributivity)} \\
&= \bigoplus_{k=1}^{D} \bigoplus_{l=1}^{D} (\mathbf{A}_{ik} \otimes \mathbf{B}_{kl}) \otimes \mathbf{C}_{lj} && \text{(associativity of } \otimes \text{ on } \mathbb{K}) \\
&= \bigoplus_{l=1}^{D} \bigoplus_{k=1}^{D} (\mathbf{A}_{ik} \otimes \mathbf{B}_{kl}) \otimes \mathbf{C}_{lj} && \text{(distributivity)} \\
&= \bigoplus_{l=1}^{D} \bigoplus_{k=1}^{D} (\mathbf{A} \otimes \mathbf{B})_{il} \otimes \mathbf{C}_{lj} && \text{(definition)} \\
&= (\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} && \text{(definition)}
\end{aligned}
$$

■

Matrices over Closed Semirings

This section introduces the extension of the Kleene closure from §2.1 to matrices and discusses the conditions needed for them to be well defined. As we will see, demonstrating the existence of a matrix asterate is deeply connected with the derivation of an efficient algorithm for its computation. Thus, the goals of this section are to present a generalization of Prop. 3.6.3 that shows we can, under some assumptions, lift a closed semiring to a closed semiring over matrices, and to provide an efficient algorithm for the computation of the Kleene star. We start with a definition.

Definition 3.6.8. *Let $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1}, {}^*)$ be a closed semiring. Consider a $D \times D$ matrix $\mathbf{M} \in \mathbb{K}^{D\times D}$. The* ***Kleene closure*** *of the matrix $\mathbf{M}$ is defined as*

$$\mathbf{M}^* \overset{\text{def}}{=} \mathbf{M}^{\otimes 0} \oplus \mathbf{M}^{\otimes 1} \oplus \mathbf{M}^{\otimes 2} \oplus \cdots = \bigoplus_{n=0}^{\infty} \mathbf{M}^{\otimes n} \tag{3.41}$$

whenever the infinite sum exists[13]. *Note we define $\mathbf{M}^0 \overset{\text{def}}{=} \mathbf{I}$ where $\mathbf{I}$ is the semiring identity matrix with elements $\mathbf{I}_{ij} \overset{\text{def}}{=} \begin{cases} \mathbf{1} & \text{if } i = j \\ \mathbf{0} & \text{otherwise} \end{cases}$ and $\mathbf{M}^{\otimes n} \overset{\text{def}}{=} \underbrace{\mathbf{M} \otimes \mathbf{M} \otimes \cdots \otimes \mathbf{M}}_{n \text{ times}}$ where $\otimes$ is matrix multiplication over the semiring $\mathcal{W}$, which, itself, involves both $\oplus$ and $\otimes$ operations, as seen in Def. 3.6.7. Moreover, we define the partial sums as*

$$\mathbf{M}^{(N)} \overset{\text{def}}{=} \bigoplus_{n=0}^{N} \mathbf{M}^{\otimes n}. \tag{3.42}$$

[13] The Kleene closure is sometimes also called the **quasi-inverse** of the matrix, for reasons that will become clear later (Gondran and Minoux, 2008).

The $\otimes$ in the exponent will usually be left out when the exact operation is clear from the context, i.e., $\mathbf{M}^{\otimes n} = \mathbf{M}^{n}$. Thus, it should be assumed that any powers are semiring powers.

We will see that the algebraic path problem as defined in the beginning of the chapter will boil down to computing the Kleene closure of a matrix over some closed semiring. In the literature, this problem is a generalization of computing the **transitive closure** of a matrix or a graph to the semiring case (Fink, 1992).

To get some intuition behind the Kleene closure, we first introduce the graph corresponding to a given matrix over a semiring.

Definition 3.6.9. *Let $\mathbf{M} \in \mathbf{M}_D(\mathcal{W})$ be a matrix over semiring $\mathcal{W}$. We define $\mathcal{G}_{\mathbf{M}}$ as the graph corresponding to the matrix $\mathbf{M}$ as the graph where the set of nodes V equals $[D]$ and any two nodes i and j are connected with a directed edge of weight $\mathbf{M}_{ij}$.*

The interpretation of the Kleene Closure. Let us now discuss the practical meaning of the Kleene closure $\mathbf{M}^{*}$. First, note that $\mathbf{M}_{ij}$, *by definition*, is the weight of the edge $i \rightarrow j$ in $\mathcal{G}_{\mathbf{M}}$. Now, we consider what $\mathbf{M}^{\otimes 2}$ means. Expanding the power, we have $\left(\mathbf{M}^{\otimes 2}\right)_{ij} = (\mathbf{M} \otimes \mathbf{M})_{ij} = \bigoplus_{k=1}^{D} \mathbf{M}_{ik} \otimes \mathbf{M}_{kj}$, i.e., we loop through all D possible nodes k in the graph, $\otimes$-multiply the weights of the the edges $i \rightarrow k$ and $k \rightarrow j$, and $\oplus$-sum those products up. We can thus think of k as intermediary nodes on a path between i and j, which shows us that $\left(\mathbf{M}^{\otimes 2}\right)_{ij}$ is the semiring sum of all paths of *exactly* length 2 from i to j. Likewise, $(\mathbf{M}^{\otimes n})_{ij}$ encodes the sum of all paths of *exactly* n from i to j. You are asked to formally prove this as an exercise at the end of the chapter. So, ${\mathbf{M}^{*}}_{ij} \overset{\text{def}}{=} \left(\bigoplus_{n=0}^{\infty} \mathbf{M}^{\otimes n}\right)_{ij}$ tells us the sum of paths of lengths $n \geq 0$ from edge i to j. It is important to note the base case of paths of length 0, i.e., $\mathbf{M}^{\otimes 0} \overset{\text{def}}{=} \mathbf{I}$, which tells us that the weight of a zero-length path is $\mathbf{0}$, unless it is a has the same source and target node, in which case it is $\mathbf{1}$. Moreover, note that this is different from a self-loop, which has the length *one* and can have any ($\mathbf{0}$ or non-$\mathbf{0}$ weight.

Existence of the Kleene Closure of a Matrix

The infinite sum in the definition of the Kleene closure of a matrix may not always exist. We now look at some special cases where the closure is guaranteed to exist.

A Generalization of the k-closed property. It turns out that we can use a straight-forward extension of the k-closed property to ensure the convergence of the infinite sum in Eq. (3.41).

Definition 3.6.10. *Let $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ be a* commutative *semiring, $\mathcal{G}$ a weighted directed graph over $\mathcal{W}$ and $k \in \mathbb{N}$. We say that $\mathcal{W}$ is **k-closed for $\mathcal{G}$** if for any* cycle *$\boldsymbol{\pi}$ in $\mathcal{G}$, it holds that*

$$\bigoplus_{n=0}^{k+1} w(\boldsymbol{\pi})^{n} = \bigoplus_{n=0}^{k} w(\boldsymbol{\pi})^{n}. \tag{3.43}$$

This is obviously a generalization of the normal k-closed property (at least for commutative semirings) in the sense that the infinite sum must only be finite for the weights of the cycles in that specific graph. If the semiring is k-closed, it is also k-closed for any weighted graph. The reason we need the semiring to be commutative in the latter case is to ensure the weight of a cycle does not depend on the node we enter it from. In the non-commutative case, the order of the multiplications of the transition weights could affect the weight of the cycle.

Note that in the case when $k = 0$, the commutativity in the definition is not required. $k = 0$ implies that for each cycle $\boldsymbol{\pi}$, $\mathbf{1} \oplus w(\boldsymbol{\pi}) = \mathbf{1}$, i.e., taking any cycle will never affect the pathsum.

Before we introduce the main results, let us define $\Pi^n(i,j)$ as the set of paths in $\mathcal{G}_{\mathbf{M}}$ between the nodes i and j which contain *exactly* n edges and $\Pi^{(n)}(i,j)$ as the set of paths between i and j containing *at most* n edges. With that, we can state the first result, which is just a reformulation of the intuition we outlined above.

Lemma 3.6.3. *The entries of the matrices* $\mathbf{M}^n$ *and* $\mathbf{M}^{(n)}$ *take the form:*

$$\left(\mathbf{M}^n\right)_{ij} = \bigoplus_{\boldsymbol{\pi} \in \Pi^n(i,j)} w\left(\boldsymbol{\pi}\right) \tag{3.44}$$

and

$$\left(\mathbf{M}^{(n)}\right)_{ij} = \bigoplus_{\boldsymbol{\pi} \in \Pi^{(n)}(i,j)} w\left(\boldsymbol{\pi}\right) \tag{3.45}$$

Proof. Outlined above and part of an exercise at the end of the chapter. ■

Lemma 3.6.4. *Let* $\mathbf{M}$ *be a* $D \times D$ *matrix over the semiring* $\mathcal{W}$*. Suppose that* $\mathcal{W}$ *is* k*-closed for* $\mathcal{G}_{\mathbf{A}}$*. Then*

$$\left(\mathbf{M}^{(n)}\right)_{ij} = \bigoplus_{\boldsymbol{\pi} \in \widetilde{\Pi^{(n)}}(i,j)} w\left(\boldsymbol{\pi}\right), \tag{3.46}$$

where $\widetilde{\Pi^{(n)}}$ *denotes the set of paths containing at most* n *edges and* traversing at most k times successively each cycle in $\mathcal{G}_{\mathbf{M}}$.

In other words, the paths containing cycles traversed more than k times successively do not add to the pathsum anymore.

Proof. We have to show that any path traversing a cycle more than k times does not need to be taken into account in Eq. (3.45).

Let $\boldsymbol{\pi}$ then be a path from i to j which successively traverses a cycle $k+q$ times for some $q \geq 1$. Let p be the node in the path where the cycle occurs. Then, $\boldsymbol{\pi}$ can be decomposed as $\boldsymbol{\pi} = \boldsymbol{\pi}(i,p)\,\boldsymbol{\pi}(p,p)^{k+q}\,\boldsymbol{\pi}(p,j)$, where $\boldsymbol{\pi}(i,p)$ is the part of the path $\boldsymbol{\pi}$ from i to p, $\boldsymbol{\pi}(p,p)$ the (single) cycle starting and ending in p and $\boldsymbol{\pi}(p,j)$ is the part of the path $\boldsymbol{\pi}$ from p to j.

If $\boldsymbol{\pi} \in \Pi^{(n)}(i,j)$ then $\boldsymbol{\pi}_r = \boldsymbol{\pi}(i,p)\,\boldsymbol{\pi}(p,p)^{r+q-1}\,\boldsymbol{\pi}(p,j) \in \Pi^{(n)}(i,j)$ as well for any $r = 0, 1, \ldots, k$.

We now show that $\boldsymbol{\pi}$ can be absorbed in Eq. (3.45) by the set of paths $\boldsymbol{\pi}_r$.

$$\begin{aligned}
w(\boldsymbol{\pi}) \oplus \bigoplus_{r=0}^{k} w(\boldsymbol{\pi}_r) = {} & w(\boldsymbol{\pi}(i,p)) \otimes w(\boldsymbol{\pi}(p,p))^{k+q} \otimes w(\boldsymbol{\pi}(p,j)) \\
& \oplus \bigoplus_{r=0}^{k} w(\boldsymbol{\pi}(i,p)) \otimes w(\boldsymbol{\pi}(p,p))^{r+q-1} \otimes w(\boldsymbol{\pi}(p,j)) \\
= {} & w(\boldsymbol{\pi}(i,p)) \otimes w(\boldsymbol{\pi}(p,p))^{q-1} \otimes \bigoplus_{r=0}^{k+1} w(\boldsymbol{\pi}(p,p))^{r} \otimes w(\boldsymbol{\pi}(p,j)) \quad \text{(distributivity)} \\
= {} & w(\boldsymbol{\pi}(i,p)) \otimes w(\boldsymbol{\pi}(p,p))^{q-1} \otimes \bigoplus_{r=0}^{k} w(\boldsymbol{\pi}(p,p))^{r} \otimes w(\boldsymbol{\pi}(p,j)) \quad (k\text{-closed}) \\
= {} & \bigoplus_{r=0}^{k} w(\boldsymbol{\pi}(i,p)) \otimes w(\boldsymbol{\pi}(p,p))^{r+q-1} \otimes w(\boldsymbol{\pi}(p,j)) \\
= {} & \bigoplus_{r=0}^{k} w(\boldsymbol{\pi}_r)
\end{aligned}$$

What this shows is that the weight of any path traversing a cycle more than k times will not add to the pathsum. This same absorption can be applied to any path successively taking a cycle more than k times. This finished the proof.

■

The next two results show how we can compute the pathsums between individual nodes in k-closed graphs. Before we introduce them, we list a few more definitions.

Definition 3.6.11. *A path $\boldsymbol{\pi}$ is **elementary** if it contains no cycle.*

Definition 3.6.12. *We further define:*

(i) the set of paths of length at most n *between the nodes i and j* traversing at most l times each elementary cycle in $\mathcal{G}_{\mathbf{M}}$ *as* $\Pi_l^{(n)}(i,j)$*;*

(ii) the set of all *paths between the nodes i and j* traversing at most l times each elementary cycle in $\mathcal{G}_{\mathbf{M}}$ *as* $\Pi_l(i,j)$*.*

*A cycle is **elementary** if it contains no other, "inner", cycles.*

Lemma 3.6.5. *If $\mathcal{W}$ is 0-closed for $\mathcal{G}_{\mathbf{M}}$, then*

$$\left(\mathbf{M}^{(n)}\right)_{ij} = \bigoplus_{\boldsymbol{\pi}\in\Pi_0^{(n)}(i,j)} w(\boldsymbol{\pi}) \tag{3.47}$$

$$\left(\mathbf{M}^{(D-1)}\right)_{ij} = \bigoplus_{\boldsymbol{\pi}\in\Pi_0(i,j)} w(\boldsymbol{\pi}) \tag{3.48}$$

Proof. Eq. (3.47) is an immediate consequence of Lemma 3.6.4 for $k = 0$. Eq. (3.48) follows Eq. (3.47) taking into account that elementary paths cannot contain more than $D-1$ edges.

■

Theorem 3.6.2. *If $\mathcal{W}$ is 0-closed for $\mathcal{G}_{\mathbf{M}}$ (without requiring the commutativity of $\mathcal{W}$), the infinite sum Eq. (3.41) has a limit, which is reached for $n \leq D-1$:*

$$\mathbf{M}^* = \mathbf{M}^{(D-1)} = \mathbf{M}^{(D)} = \cdots \tag{3.49}$$

Moreover, $\mathbf{M}^*$ satisfies the equation $\mathbf{M}^* = \mathbf{I} \oplus \mathbf{M}^* \otimes \mathbf{M} = \mathbf{I} \oplus \mathbf{M} \otimes \mathbf{M}^*$.

Proof. The finite limit is a direct consequence of Lemma 3.6.5. We verify that $\mathbf{M}^*$ is a solution to the equations above:

$$\begin{aligned}
\mathbf{I} \oplus \mathbf{M}^* \otimes \mathbf{M} &= \mathbf{I} \oplus \left(\mathbf{I} \oplus \mathbf{M} \oplus \ldots \mathbf{M}^{(D-1)}\right) \otimes \mathbf{M} && \text{(0-closed)} && (3.50)\\
&= \mathbf{I} \oplus \mathbf{M} \oplus \ldots \mathbf{M}^{(D)} && \text{(distributivity)} && (3.51)\\
&= \mathbf{I} \oplus \mathbf{M} \oplus \ldots \mathbf{M}^{(D-1)} && \text{(0-closed)} && (3.52)\\
&= \mathbf{M}^* && \text{(0-closed)} && (3.53)
\end{aligned}$$

The second equation can be verified the same way. ■

As we can see, Thm. 3.6.2 does not require a commutative semiring. However, the generalization to $l > 0$ requires us to make this assumption, as shown in the next theorem.

Theorem 3.6.3. *If $\mathcal{W}$ is k-closed for $\mathcal{G}_{\mathbf{M}}$ (with $\mathcal{W}$ being commutative), then*

$$\left(\mathbf{M}^{(n)}\right)_{ij} = \bigoplus_{\boldsymbol{\pi} \in \Pi_k^{(n)}(i,j)} w\left(\boldsymbol{\pi}\right) \tag{3.54}$$

$$\left(\mathbf{M}^{(D_k)}\right)_{ij} = \bigoplus_{\boldsymbol{\pi} \in \Pi_k(i,j)} w\left(\boldsymbol{\pi}\right), \tag{3.55}$$

where D_k is the maximum number of edges of the paths in $\Pi_k\left(i,j\right)$. *Furthermore, $D_k \leq D-1+kDt$, where t is the total number of elementary cycles in $\mathcal{G}_{\mathbf{M}}$. Moreover, $\mathbf{M}^*$ satisfies the equation $\mathbf{M}^* = \mathbf{I} \oplus \mathbf{M}^* \otimes \mathbf{M} = \mathbf{I} \oplus \mathbf{M} \otimes \mathbf{M}^*$.*

Proof. First, notice that the number of edges in any path in $\Pi_l\left(i,j\right)$ is limited by $D-1+lDt$ where t marks the total number of elementary cycles in $\mathcal{G}_{\mathbf{A}}$. The term $D-1$ comes from the D nodes in the graph while the second from the limited number (l) of repetitions of any of the t cycles of at most D nodes.

Eq. (3.54) follows from Lemma 3.6.3. Due to commutativity, the weight of any cycle is the same no matter where it is entered. This means that all traversals can be "grouped together" and the k-closed property can be applied.

Eq. (3.55) can be shown by noticing that if a path is longer than D_k, then, by definition, there will be an elementary cycle in the path of which has been traversed more than k times, meaning that it can be absorbed as in the proof for Lemma 3.6.4.

The fact that $\mathbf{M}^*$ is a solution to the equations is shown the same way as in Thm. 3.6.2. ■

The real semiring. Note that none of the theorems above apply to the case of the real semiring. To characterize the existence of the Kleene closure in that case, we can analyze the behavior of the infinite sum Eq. (3.41). Since the algebraic structure $(\mathbb{R}, +, \cdot, 0, 1)$ is in fact a field, we can use (additive and multiplicative) inverses to characterize it.

Let $\mathbf{M}$ be a $D \times D$ matrix. Suppose that the infinite sum Eq. (3.41) exists. It is easy to show that it then satisfies the equation

$$\mathbf{M}^* = \mathbf{I} + \mathbf{M}\mathbf{M}^*, \tag{3.56}$$

which we can rewrite as

$$\mathbf{M}^* = (\mathbf{I} - \mathbf{M})^{-1}, \tag{3.57}$$

giving us the result that $\mathbf{M}^* = (\mathbf{I} - \mathbf{M})^{-1}$. Note that the inverse of $\mathbf{I} - \mathbf{M}$ necessarily exists, since the convergence of the infinite sum implies that, for all eigenvalues λ_i of $\mathbf{M}$, it holds that $|\lambda_i| < 1$.

We will in fact generalize this to any matrix $\mathbf{M}$ such that $\mathbf{I} - \mathbf{M}$ is nonsingular at the end of the section. However, the interpretation of the closure as the infinite sum is only applicable to the case when the infinite sum actually exists.

3.6.5 A Fixed-Point Algorithm

We will now develop a first attempt at computing the Kleene closure. This is the so-called fixed-point algorithm, which $\oplus$-sums matrix powers of $\mathbf{M}$ until the elements of the sum stop changing—hence, the name fixed-point. The algorithm is presented in Alg. 7.

Algorithm 7 The FixedPoint algorithm.

```
def FixedPoint($\mathbf{M}$):
    $\mathbf{M}^{(0)} \leftarrow \mathbf{I}$                         ▷ Initialize with a matrix of all 0
    repeat
        $\mathbf{M}^{(n)} \leftarrow \mathbf{I} \oplus \mathbf{M}^{(n-1)} \otimes \mathbf{M}$
    until the entries of $\mathbf{M}^{(n)}$ stop changing.
    return $\mathbf{M}^{(n)}$
```

Using simple induction, we can see that the matrix $\mathbf{M}^{(n)}$ represents the sum $\bigoplus_{j=0}^{n} \mathbf{M}^j$, meaning that the algorithm returns $\bigoplus_{n=0}^{\infty} \mathbf{M}^n$, whenever it finishes.

In the general case, however, Alg. 7 is not exact, as the infinite sum might in fact be infinite. This might be the case for semirings which are not k-closed, for example the real semiring $(\mathbb{R}, +, \times, 0, 1)$. In those cases, FixedPoint must be iterated until numerical convergence (which is obviously not guaranteed either, as the iterates may diverge). However, as a direct consequence of the existence theorems from the previous sections, the algorithm is exact for the important special cases of k-closed semirings, defined above, the Alg. 7 is exact. A natural consequence of Thm. 3.6.3 is that, for 0-closed semirings, Alg. 7 terminates in exactly D iterations. Since each semiring matrix multiplication takes $\mathcal{O}\left(D^3\right)$ time[14], the entire procedure takes $\mathcal{O}\left(D^4\right)$ time. Finally, despite being a relatively trivial consequence of the k-closed property, we wrap the result in the theorem as a mark of distinction.

Theorem 3.6.4. *The FixedPoint algorithm exactly computes the Kleene closure of a $D \times D$ matrix over a 0-closed semiring in $\mathcal{O}\left(D^4\right)$ time. Furthermore, FixedPoint exactly computes the Kleene closure of a $D \times D$ matrix over a k-closed semiring in $\mathcal{O}\left((D - 1 + ktD)D^3\right)$ time, where t is the number of elementary cycles in the matrix-induced graph.*

Improving the Quartic Runtime. With a bit more care, we can reduce the runtime of this fixed point procedure to $\mathcal{O}\left(D^3 \log D\right)$. This follows from a relatively simple observation. To start with a simpler problem, consider a real number $r \in \mathbb{R}$ and a positive integer m. Suppose we wish to compute r^m. Naïvely, this would take m multiplications:

$$r^m \overset{\text{def}}{=} \underbrace{r \times r \cdots r}_{m \text{ times}} \tag{3.58}$$

If we decompose the integer m into a sum of powers of two (which itself can be done in $\mathcal{O}(\log_2 m)$ time using the same procedure as converting the number to binary), we quickly see that we only need $\mathcal{O}(\log_2 m)$ operations: Consider the case of $m = 21$. We have $21 = \underbrace{1}_{2^0} + \underbrace{4}_{2^2} + \underbrace{16}_{2^4}$. More generally, every positive integer can be decomposed into a sum of at most $\log_2 m$ distinct powers of 2. This gives us a speed-up because we can compute the powers of with the following procedure:

$r^0 \leftarrow 1$

[14] Faster matrix multiplication algorithm exist for real matrices, but they are generally not applicable to matrices over weaker semirings. We do not consider any such algorithms in this course.

```
for $j = 1, \ldots, \lfloor \log_2 m \rfloor$ :
    $r^{2^j} \leftarrow r^{2^{j-1}} \times r^{2^{j-1}}$
```

We then just multiply together the powers of r^{2^j} for those j which make up the decomposition of m. This *same* algorithm can be applied to computing $\mathbf{M}^{\otimes d}$ for $d \in \{1, \ldots, D\}$:

```
$\mathbf{M}^0 \leftarrow \mathbf{I}$
for $j = 1, \ldots, \lfloor \log_2 m \rfloor$ :
    $\mathbf{M}^{2^j} \leftarrow \mathbf{M}^{2^{j-1}} \otimes \mathbf{M}^{2^{j-1}}$
```

Since we loop through at most $\lfloor \log_2 m \rfloor$ iterations and each step takes $\mathcal{O}\left(D^3\right)$, the runtime of this algorithm is $\mathcal{O}\left(D^3 \log_2 D\right)$. Note that we have made use of the associativity property of the multiplication operation in deriving this algorithm.

You might wonder how this construction of matrix powers helps when computing the closure with the fixed-point algorithm, since we need to sum *all* the matrix powers from 0 to N, not just the exponentially-spaced ones. The following proposition tells us exactly why.

Proposition 3.6.4. *Let* $\mathbf{A}$ *be a* $D \times D$ *matrix over the semiring* $\mathcal{W}$. *If* $\mathcal{W}$ *is idempotent, then*

$$\mathbf{A}^{(N)} = \left(\mathbf{I} \oplus \mathbf{A}\right)^N . \tag{3.59}$$

Proof. Using the binomial theorem, we can calculate

$$\left(\mathbf{I} \oplus \mathbf{A}\right)^N = \bigoplus_{n=0}^{N} \binom{N}{n} \mathbf{A}^n \tag{3.60}$$

$$= \bigoplus_{n=0}^{N} \mathbf{A}^n \qquad \text{(idempotency)} \tag{3.61}$$

$$= \mathbf{A}^{(N)} \qquad \text{(definition)} \tag{3.62}$$

Note that here $c\mathbf{M} = \underbrace{\mathbf{M} \oplus \ldots \oplus \mathbf{M}}_{\text{c times}}$. ■

This means that we can use the sped-up algorithm for matrix powers on $(\mathbf{I} \oplus \mathbf{A})$ directly to get the closure.

3.6.6 Lehmann's Algorithm

This section introduces the main algorithm of the chapter which can be used to exactly calculate the Kleene closure of a matrix over *any* closed semiring, as long as the closure exists. It is very similar to the Gauss–Jordan elimination algorithm for computing the inverse of a matrix, and can be seen as a generalization of the well-known Floyd–Warshall algorithm for computing the closure of a matrix to an arbitrary closed semiring.

We will first take a closer look at what conditions we want the closure to satisfy and then, through some algebraic manipulations and the use of the Kleene operator from the semiring, come up with the algorithm for computing the closure of the matrix.

The Algorithm and Intuition

Lehmann's algorithm is a very general dynamic program for computing the Kleene closure of matrices over arbitrary closed semirings. While the formal derivation and proof of correctness is relatively involved, the algorithm itself and intuition behind it are surprisingly simple. Therefore, we will first devote a few words to presenting and explaining the algorithm on the intuitive level before deriving it and proving its correctness.

The algorithm computes the $\oplus$-sum over the paths between any two nodes in a graph with the weighted adjacency matrix $\mathbf{M}$, i.e.,

$$\mathbf{R}_{ik} \overset{\text{def}}{=} \bigoplus_{\boldsymbol{\pi} \in \Pi(\mathcal{A})(i,k)} w_{\mathrm{I}}\left(\boldsymbol{\pi}\right) \tag{3.63}$$

In graph theory, this is known as the all-pairs shortest-path problem. These D^2 entries can later be used to compute the pathsum in a cyclic WFSA $\mathcal{A}$:

$$\mathrm{Z}\left(\mathcal{A}\right) \overset{\text{def}}{=} \bigoplus_{i,k \in Q} \lambda\left(q_i\right) \otimes \mathbf{R}_{ik} \otimes \rho\left(q_k\right) \tag{3.64}$$

The intuition behind the algorithm will be very familiar if you have seen the Floyd–Warshall algorithm before. The idea is to *divide* the set of paths between any two states i and k $\Pi\left(i,k\right)$ into subsets of paths as follows. We denote with $\Pi^{\leq j}\left(i,k\right)$ the set of paths $\boldsymbol{\pi} \in \Pi\left(i,k\right)$ *which do not cross any nodes with indices* $> j$. As we will see, this allows for a convenient partition of the entire set of paths $\Pi\left(i,k\right)$. We will use the notation $\mathbf{R}_{ik}$ to denote the pathsum of the paths in $\Pi\left(i,k\right)$ and $\mathbf{R}_{ik}^{\leq m}$ to denote the pathsum of the paths in $\Pi^{\leq j}\left(i,k\right)$. The crucial observation is that any path $\boldsymbol{\pi} \in \Pi^{\leq j}\left(i,k\right)$ either *crosses* j or it *does not*. In the latter case, $\boldsymbol{\pi} \in \Pi^{\leq j-1}\left(i,k\right)$. In the former, however, we can decompose $\boldsymbol{\pi}$ into a cycle which starts and ends in j (the cycle can be of length 0), the part *before* the cycle and the part *after* it. Similarly to above, we denote this with $\boldsymbol{\pi} = \boldsymbol{\pi}_{ij}\boldsymbol{\pi}_{jj}\boldsymbol{\pi}_{jk}$. Intuitively, this decomposition together with distributivity allow us to also decompose the pathsum $\mathbf{R}_{ik}$ into $\mathbf{R}_{ij}^{\leq j-1} \otimes \mathbf{R}_{jj}^{\leq j-1} \otimes \mathbf{R}_{jk}^{\leq j-1}$. This of course only takes into account the paths which actually cross j. The pathsum of the paths which *do not cross* j, on the other hand, is $\mathbf{R}_{ik}^{\leq j-1}$. Obviously, $\mathbf{R}_{ik}^{\leq D} = \mathbf{R}_{ik}$. This defines a natural procedure to build up the pathsum from the partial pathsums $\mathbf{R}_{ik}^{\leq j}$ as

$$\mathbf{R}_{ik}^{\leq j} \leftarrow \mathbf{R}_{ik}^{\leq j-1} \oplus \mathbf{R}_{ij}^{\leq j-1} \otimes \mathbf{R}_{jj}^{\leq j-1} \otimes \mathbf{R}_{jk}^{\leq j-1} \tag{3.65}$$

All that is left are the starting conditions, which are defined as $\mathbf{R}^{\leq 0} \overset{\text{def}}{=} \mathbf{M}$, as the values in $\mathbf{M}$ capture the paths not passing through any intermediary nodes.[15] The algorithm is outlined in Alg. 8.

Algorithm 8 Lehmann's algorithm

```
1. def Lehmann($\mathbf{M}$):
2.   ▷ $\mathbf{M}$ is a $D \times D$ matrix over a closed semiring
3.   $\mathbf{R}^{(0)} \leftarrow \mathbf{M}$
4.   for $j \leftarrow 1$ up to $D$ :
5.     for $i \leftarrow 1$ up to $D$ :
6.       for $k \leftarrow 1$ up to $D$ :
7.         $\mathbf{R}_{ik}^{(j)} \leftarrow \mathbf{R}_{ik}^{(j-1)} \oplus \mathbf{R}_{ij}^{(j-1)} \otimes \left(\mathbf{R}_{jj}^{(j-1)}\right)^{*} \otimes \mathbf{R}_{jk}^{(j-1)}$
8.   return $\mathbf{I} \oplus \mathbf{R}^{(D)}$
```

[15] Technically, repeated self loops do not pass through any nodes either. However, this is captured in the update rule on Line 7 of the algorithm for $j = 1$, as the values of $\mathbf{R}^{(1)}$ as updated with $(\mathbf{R}_{jj}^{(0)})^{*}$.

Example 3.6.6. *Consider the following cyclic WFSA.*

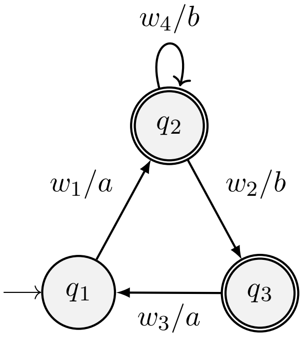


Figure 3.17: A cyclic WFSA $\mathcal{A}$.

Firstly, we derive the adjacency matrix of $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ *as*

$$M(\mathcal{A}) \stackrel{\text{def}}{=} \bigoplus_{a \in \Sigma} M^{(a)}.$$

We will remove the dependence on $\mathcal{A}$ *from now on and we will only refer to the summarized WFSA graph and its associated adjacency matrix* M*. In our example WFSA, the adjacency matrix* M *can be computed as follows*

$$M^{(a)} = \begin{bmatrix} \boldsymbol{0} & w_1 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ w_3 & \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \quad M^{(b)} = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & w_4 & w_2 \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \quad M = M^{(a)} \oplus M^{(b)} = \begin{bmatrix} \boldsymbol{0} & w_1 & \boldsymbol{0} \\ \boldsymbol{0} & w_4 & w_2 \\ w_3 & \boldsymbol{0} & \boldsymbol{0} \end{bmatrix}.$$

Suppose that

$$\lambda(q_1) \stackrel{\text{def}}{=} w_{I,1}$$
$$\lambda(q_2) = \lambda(q_3) \stackrel{\text{def}}{=} \boldsymbol{0}$$

and

$$\rho(q_1) \stackrel{\text{def}}{=} \boldsymbol{0}$$
$$\rho(q_2) \stackrel{\text{def}}{=} w_{F,2}$$
$$\rho(q_3) \stackrel{\text{def}}{=} w_{F,3}$$

Then, according to Eq. (3.64), the all-pair pathsum of the WFSA equals to

$$Z(\mathcal{A}) = w_{I,1} \otimes \mathbf{R}_{12} \otimes w_{F,2} \oplus w_{I,1} \otimes \mathbf{R}_{13} \otimes w_{F,3}.$$

Lehmann's algorithm computes the matrix $\mathbf{R}$ *iteratively, as presented in Alg. 8. In this example, the number of nodes in the WFSA is equal to* $D = 3$*. We start by initializing the matrix* $\mathbf{R}^{(0)}$*, which is by definition equal to* $\mathbf{M}$ *(Line 3). In the first iteration, i.e. for* $j = 1$*, we update all the entries of bR (Lines 5, 6, 7) and get the following resulting matrix*

$$\mathbf{R}^{(1)} = \begin{bmatrix} \boldsymbol{0} & w_1 & \boldsymbol{0} \\ \boldsymbol{0} & w_4 & w_2 \\ w_3 & w_3 \otimes w_1 & \boldsymbol{0} \end{bmatrix}.$$

In the second iteration, i.e. for $j = 2$, the matrix $\mathbf{R}$ becomes

$$\mathbf{R}^{(2)} = \begin{bmatrix} \boldsymbol{0} & w_1 \oplus w_1 \otimes w_4^* \otimes w_4 & w_1 \otimes w_4^* \otimes w_2 \\ \boldsymbol{0} & w_4 \oplus w_4 \otimes w_4^* \otimes w_4 & w_2 \oplus w_4 \otimes w_4^* \otimes w_2 \\ w_3 & w_3 \otimes w_1 \oplus w_3 \otimes w_1 \otimes w_4^* \otimes w_4 & w_3 \otimes w_1 \otimes w_4^* \otimes w_2 \end{bmatrix}$$

For the last iteration, i.e. for $j = 3$, for the sake of simplicity we only report the results for the paths whose initial and final weights are non-zero, that is those paths which will be actually used in the computation of the pathsum

$$\begin{aligned} \mathbf{R}_{12}^{(3)} &= w_1 \oplus w_1 \otimes w_4^* \otimes w_4 \\ &\quad \oplus (w_1 \otimes w_4^* \otimes w_2) \otimes (w_3 \otimes w_1 \otimes w_4^* \otimes w_2)^* \otimes (w_3 \otimes w_1 \oplus w_3 \otimes w_1 \otimes w_4^* \otimes w_4) \\ \mathbf{R}_{13}^{(3)} &= w_1 \otimes w_4^* \otimes w_2 \\ &\quad \oplus (w_1 \otimes w_4^* \otimes w_2) \otimes (w_3 \otimes w_1 \otimes w_4^* \otimes w_2)^* \otimes (w_3 \otimes w_1 \otimes w_4^* \otimes w_2) \end{aligned}$$

Finally, we $\oplus$-sum to $\mathbf{R}^{(3)}$ the matrix $\mathbf{I}$ (Line 8) and get the values $\mathbf{R}_{12}$ and $\mathbf{R}_{13}$ needed to compute the pathsum $Z(\mathcal{A})$.

Notable Special Cases

One of the most beautiful aspects of Lehmann's algorithm is that it generalizes many well known algorithms. In this section, we discuss three cases.

Floyd–Warshall. The Floyd–Warshall algorithm is a well known algorithm for the **all-pairs shortest paths problem**, i.e., the task of finding the shortest distance between any pair of nodes in a graph where the weights of the edges in a path are additive. It works on any graph without negative cycles, since in that case, the shortest paths are undefined (we could always loop in the cycle to lower the weight of a path).

The intuition behind the algorithm is exactly the same as in our reasoning in §3.6.4. However, instead of *summing* the path weights between two nodes, we are interested in the *minimum* over them. This naturally applies to the **tropical semiring**. Indeed, the Floyd–Warshall algorithm in its original form is exactly the Lehmann algorithm over the tropical semiring! Recall that in the 0-closed tropical semiring, the Kleene operator is given by $x^* = 0, \forall x$, meaning that the expression on Line 7 simplifies. The algorithm with the specific tropical semiring operations is presented in Alg. 9.

Algorithm 9 The FloydWarshall algorithm.

```
1. def FloydWarshall($\mathbf{M}$):
2.   ▷ $\mathbf{M}$ is a $D \times D$ matrix over a closed semiring
3.   $\mathbf{R}^{(0)} \leftarrow \mathbf{M}$
4.   for $j \leftarrow 1$ up to $D$ :
5.     for $i \leftarrow 1$ up to $D$ :
6.       for $k \leftarrow 1$ up to $D$ :
7.         $\mathbf{R}_{ik}^{(j)} \leftarrow \min\left(\mathbf{R}_{ik}^{(j-1)}, \mathbf{R}_{ij}^{(j-1)} + \mathbf{R}_{jk}^{(j-1)}\right)$
8.   return $\min\left(\mathbf{I}, \mathbf{R}^{(D)}\right)$        ▷ $\mathbf{I}$ is the matrix with zeros on the diagonal and $\infty$ elsewhere.
```

Alternatively, the same name is sometimes also given to the algorithm for finding the **transitive closure** of a graph. The name transitive closure comes from the interpretation of a graph as a representation of a binary relation R. The transitive closure R^+ of a relation R is, in words, the smallest relation that contains R and is transitive. A relation S is **transitive** if, for any x, y, z, $xSy \wedge ySz \implies xSz$, where we use the relation S with infix notation.

Let us now define the transitive closure of a graph formally.

Definition 3.6.13. *Let $\mathcal{G}$ be a graph. The* ***transitive closure*** *of $\mathcal{G}$ is the matrix $\mathbf{T}$ with the entries*

$$\mathbf{T}_{ij} = \mathbb{1}\left\{\textit{There exists a directed path } \boldsymbol{\pi} \textit{ from } i \textit{ to } j\right\}. \tag{3.66}$$

In this version of the algorithm, we are again not “summing” over the paths between two nodes, instead, we are only interested in their *existance*. This naturally calls for the **Boolean semiring**. Like the tropical semiring above, Boolean semiring is also 0-closed, meaning that the Kleene operator is given by $x^* = 1, \forall x$. Again, the expression on Line 7 simplifies. The algorithm with the specific Boolean semiring operations is presented in Alg. 10.

Algorithm 10 The FloydWarshall algorithm for the transitive closure.

```
1. def FloydWarshall($\mathbf{M}$):
2.   ▷ $\mathbf{M}$ is a $D \times D$ matrix over a closed semiring
3.   $\mathbf{R}^{(0)} \leftarrow \mathbf{M}$
4.   for $j \leftarrow 1$ up to $D$ :
5.     for $i \leftarrow 1$ up to $D$ :
6.       for $k \leftarrow 1$ up to $D$ :
7.         $\mathbf{R}^{(j)}_{ik} \leftarrow \mathbf{R}^{(j-1)}_{ik} \vee \mathbf{R}^{(j-1)}_{ij} \wedge \mathbf{R}^{(j-1)}_{jk}$
8.   return $\mathbf{I} \vee \mathbf{R}^{(D)}$
```

Gauss–Jordan. To further showcase the generality of Lehmann’s algorithm, let us see how it can be applied to the problem of matrix inversion for matrices over the real field $(\mathbb{R}, +, \cdot, 0, 1)$. The fact we are operating over a field rather than the weaker semiring means that we can define additional operations and that we can use more guarantees on concepts such as uniqueness of solutions to equations. Indeed, field give us subtraction (additive inverses) and division (multiplicative inverses) by definition.

Let $\mathbf{M} \in \mathbb{R}^{D \times D}$ be a matrix. We consider the problem of finding the inverse $(\mathbf{I} - \mathbf{M})^{-1}$.

Whenever the Kleene closure of the matrix $\mathbf{M}$, $\mathbf{M}^*$, exists, it satisfies $\mathbf{M}^* = \mathbf{I} + \mathbf{M}\mathbf{M}^*$. We rearrange this to

$$\mathbf{M}^* = \mathbf{I} + \mathbf{M}\mathbf{M}^* \tag{3.67}$$

$$\mathbf{M}^* - \mathbf{M}\mathbf{M}^* = \mathbf{I} \tag{3.68}$$

$$\mathbf{M}^* (\mathbf{I} - \mathbf{M}) = \mathbf{I} \tag{3.69}$$

$$\mathbf{M}^* = (\mathbf{I} - \mathbf{M})^{-1}. \tag{3.70}$$

Therefore, analogously to the Kleene closure on the field of reals itself, the Kleene closure of a matrix $\mathbf{M}$ is the inverse of $\mathbf{I} - \mathbf{M}$! This is then also what Lehmann’s algorithm computes for a given matrix $\mathbf{M}$.

Normally, however, we are not interested in $(\mathbf{I}-\mathbf{M})^{-1}$ for a given $\mathbf{M}$, but rather directly in $\mathbf{M}^{-1}$. We can easily use Lehmann to compute it as well, by computing the closure of the matrix $\mathbf{I}-\mathbf{M}$:

$$(\mathbf{I}-\mathbf{M})^{*} = \mathbf{I} + (\mathbf{I}-\mathbf{M})(\mathbf{I}-\mathbf{M})^{*} \tag{3.71}$$

$$(\mathbf{I}-\mathbf{M})^{*} - (\mathbf{I}-\mathbf{M})(\mathbf{I}-\mathbf{M})^{*} = \mathbf{I} \tag{3.72}$$

$$(\mathbf{I}-\mathbf{M})^{*}(\mathbf{I}-(\mathbf{I}-\mathbf{M})) = \mathbf{I} \tag{3.73}$$

$$(\mathbf{I}-\mathbf{M})^{*} = \mathbf{M}^{-1}. \tag{3.74}$$

Kleene's Algorithm. Kleene's algorithm is an algorithm for transforming a weighted finite state automaton into a (weighted) regular expression. Thus, together with Thompson's construction of a weighted finite state automaton from a weighted regular expression, it establishes the equivalence of weighted finite state automata and weighted regular expressions. We will cover the algorithm in detail in §3.12.1. Here, we just note that it is also another special instance of Lehmann's algorithm over a specific semiring, called Kleene's semiring. We postpone a detailed description of the algorithm and the semiring over which it operates to §3.12.1.

Derivation and Correctness

This section derives and shows the correctness of Lehmann over semirings which are not fields but satisfy some other properties, such as having an order relation. As we will see, the properties of k-closeness will also play a major part in the correctness of the algorithm.

Our approach to computing the Kleene star of a matrix $\mathbf{M}$ is by constructing the solution to the following equation

$$\mathbf{Y} = \mathbf{Y} \otimes \mathbf{M} \oplus \mathbf{I}. \tag{3.75}$$

We will see that under some assumptions, the solution we will construct exactly corresponds the Kleene closure of $\mathbf{M}$. These assumptions, from a high level, will guarantee that the infinite sum $\mathbf{M}^{*}$ exists and that it, too, is a solution to Eq. (3.75). We will then be able to show that the solution found by our construction coincides with the Kleene closure.

To construct a solution to Eq. (3.75), however, it is easier to solve a slightly more general equation

$$\mathbf{Y} = \mathbf{Y} \otimes \mathbf{M} \oplus \mathbf{B} \tag{3.76}$$

where we have replaced the $\mathbf{I}$ with a $\mathbf{B}$. Note that we only consider one of the two axiomatic properties of the Kleene closure. The other one is $\mathbf{Y} = \mathbf{M} \otimes \mathbf{Y} \oplus \mathbf{B}$

The general approach we will take to solving this equations is one of **elimination**. We will first express the solution $\mathbf{Y}$ in terms of the *last* $(D-1)$ *columns*, i.e., the columns numbered $2, \ldots, D$ of $\mathbf{Y}$. This, in effect, eliminates, the first column of $\mathbf{Y}$ from the solution. We will continue in a fashion, expressing $\mathbf{Y}$ in terms of the *last* $(D-k)$ *columns of* $\mathbf{Y}$, i.e., the columns numbered $k, \ldots, D$ until we have eliminated all columns, in which case we have expressed the solution $\mathbf{Y}$ in terms of a quantity other than $\mathbf{Y}$ itself.

We will now prove a simple proposition that will be used as the base case of a general recurrence relationship.

Proposition 3.6.5. *Eq. Eq.* (3.76) *may be re-written as*

$$\mathbf{Y} = \mathbf{Y}^{(1)} \otimes \mathbf{M}^{(1)} \oplus \mathbf{B}^{(1)} \tag{3.77}$$

where we define $\mathbf{Y}^{(\ell)}$ *as the matrix obtained by setting the first* ℓ *columns of* $\mathbf{Y}$ *to* $\mathbf{0}$ *and the matrices*

$$\mathbf{M}^{(1)}_{jk} \overset{\text{def}}{=} \mathbf{M}_{jk} \oplus \mathbf{M}_{j1} \otimes \mathbf{M}^{*}_{11} \otimes \mathbf{M}_{1k} \tag{3.78}$$

$$\mathbf{B}^{(1)}_{jk} \overset{\text{def}}{=} \mathbf{B}_{jk} \oplus \mathbf{B}_{j1} \otimes \mathbf{M}^{*}_{11} \otimes \mathbf{M}_{1k} \tag{3.79}$$

Proof. On one hand, due to a result proved in the Appendix, we have

$$\mathbf{Y}_{i1} = \bigoplus_{j=2}^{D} \mathbf{Y}_{1j} \otimes \mathbf{M}_{j1} \otimes \mathbf{M}^{*}_{11} \oplus \mathbf{B}_{i1} \otimes \mathbf{M}^{*}_{11} \qquad \text{(Lemma 3.6.7)} \tag{3.80}$$

And, on the other, using Eq. (3.76), we can now express $\mathbf{Y}_{ik}$ for any i and $k \geq 2$ as:

$$\mathbf{Y}_{ik} = \bigoplus_{j=1}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{jk} \oplus \mathbf{B}_{ik} \tag{3.81}$$

$$= \mathbf{Y}_{i1} \otimes \mathbf{M}_{1i} \oplus \bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{jk} \oplus \mathbf{B}_{ik} \qquad \text{(pulling out first term)} \tag{3.82}$$

Next, we plug Eq. (3.80) into Eq. (3.82) and simplify algebraically. The algebraic manipulation is shown below step by step:

$$\begin{aligned}
\mathbf{Y}_{ik} &= \mathbf{Y}_{i1} \otimes \mathbf{M}_{1i} \oplus \bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{jk} \oplus \mathbf{B}_{ik} \\
&= \left(\bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{j1} \otimes \mathbf{M}^{*}_{i1} \oplus \mathbf{B}_{i1} \otimes \mathbf{M}^{*}_{i1} \right) \otimes \mathbf{M}_{1i} \oplus \bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{jk} \oplus \mathbf{B}_{ik} && \text{(Plug in Eq. (3.80))} \\
&= \bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{j1} \otimes \mathbf{M}^{*}_{i1} \otimes \mathbf{M}_{1i} \oplus \mathbf{B}_{i1} \otimes \mathbf{M}^{*}_{i1} \otimes \mathbf{M}_{1i} \oplus \bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{jk} \oplus \mathbf{B}_{ik} && \text{(distributivity)} \\
&= \bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{j1} \otimes \mathbf{M}^{*}_{i1} \otimes \mathbf{M}_{1i} \oplus \bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{jk} \oplus \mathbf{B}_{i1} \otimes \mathbf{M}^{*}_{i1} \otimes \mathbf{M}_{1i} \oplus \mathbf{B}_{ik} && \text{(commutativity of } \oplus\text{)} \\
&= \bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes (\mathbf{M}_{j1} \otimes \mathbf{M}^{*}_{i1} \otimes \mathbf{M}_{1i} \oplus \mathbf{M}_{jk}) \oplus \mathbf{B}_{i1} \otimes \mathbf{M}^{*}_{i1} \otimes \mathbf{M}_{1i} \oplus \mathbf{B}_{ik} && \text{(distributivity)}
\end{aligned} \tag{3.83}$$

Substituting the definitions Eq. (3.78) and Eq. (3.79) into Eq. (3.83) yields

$$\mathbf{Y}_{ik} = \bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}^{(1)}_{jk} \oplus \mathbf{B}^{(1)}_{ik} \tag{3.84}$$

$$= \bigoplus_{j=1}^{D} \mathbf{Y}^{(1)}_{ij} \otimes \mathbf{M}^{(1)}_{jk} \oplus \mathbf{B}^{(1)}_{ik} \tag{3.85}$$

Now, rewriting Eq. (3.84) as a matrix equation gives the desired result. ■

To reiterate, it is important to realize what we have done with this—we have expressed the *first column* of the solution to Eq. (3.76) in terms of the other columns. The idea of the rest of the derivation is to successively express more and more columns this way. Indeed, it is this rewriting that makes our method analogous to Gaussian elimination in that we eliminate the columns—one at a time. We will now generalize Prop. 3.6.5 to the case of ℓ in Thm. 3.6.5, but before we do that, we introduce a few matrices which will simplify our notation. These generalize the matrices defined above.

First, we define the matrix $\mathbf{M}^{(0)}$ as

$$\mathbf{M}^{(0)} = \mathbf{M} \tag{3.86}$$

and then mutually recursively define the matrices $\mathbf{M}^{(\ell)}$ and $\mathbf{G}^{(\ell)}$ as follows.

Definition 3.6.14. *For any $\ell \in \{1, \dots, D\}$, we define the matrices $\mathbf{G}^{(\ell)}$ as*

$$\mathbf{G}^{(\ell)}_{ik} \overset{\text{def}}{=} \begin{cases} \left(\mathbf{M}^{(\ell-1)}_{\ell\ell}\right)^* & \textbf{\textit{if}}\ i = k = \ell \\ \left(\mathbf{M}^{(\ell-1)}_{\ell\ell}\right)^* \otimes \mathbf{M}^{(\ell-1)}_{\ell k} & \textbf{\textit{if}}\ i = \ell\ \textbf{\textit{and}}\ k \neq \ell \\ \delta_{ik} & \textbf{\textit{otherwise}} \end{cases} \tag{3.87}$$

where δ_{ij} is the semiring Dirac δ function. We can visualize $\mathbf{G}^{(\ell)}$ as follows:

$$\mathbf{G}^{(\ell)} = \begin{array}{cc} \begin{array}{cccccc} 1 & 2 & & \ell & & D \end{array} & \\ \left(\begin{array}{cccccc} \boldsymbol{1} & \boldsymbol{0} & & \cdots & & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{1} & & \cdots & & \boldsymbol{0} \\ \vdots & \vdots & & \vdots & & \vdots \\ \left(\mathbf{M}^{(\ell-1)}_{\ell\ell}\right)^* \otimes \mathbf{M}^{(\ell-1)}_{\ell 1} & \left(\mathbf{M}^{(\ell-1)}_{\ell\ell}\right)^* \otimes \mathbf{M}^{(\ell-1)}_{\ell 2} & \cdots & \left(\mathbf{M}^{(\ell-1)}_{\ell\ell}\right)^* & \cdots & \left(\mathbf{M}^{(\ell-1)}_{\ell\ell}\right)^* \otimes \mathbf{M}^{(\ell-1)}_{\ell D} \\ \vdots & \vdots & & \vdots & & \vdots \\ \boldsymbol{0} & \boldsymbol{0} & & \cdots & & \boldsymbol{1} \end{array}\right) & \begin{array}{c} 1 \\ 2 \\ \\ \ell \\ \\ D \end{array} \end{array} \tag{3.88}$$

Furthermore, we define the matrices

$$\mathbf{M}^{(\ell)} \overset{\text{def}}{=} \mathbf{M}^{(\ell-1)} \otimes \mathbf{G}^{(\ell)} \tag{3.89}$$

$$\mathbf{B}^{(\ell)} \overset{\text{def}}{=} \mathbf{B}^{(\ell-1)} \otimes \mathbf{G}^{(\ell)} \tag{3.90}$$

Note that the above definitions of $\mathbf{M}^{(j)}$ and $\mathbf{B}^{(j)}$ are equivalent to the ones in Prop. 3.6.5, but generalized to any j.

Theorem 3.6.5. *The equation*

$$\mathbf{Y} = \mathbf{Y}^{(\ell)} \otimes \mathbf{M}^{(\ell)} \oplus \mathbf{B}^{(\ell)} \tag{3.91}$$

holds for all $\ell \in \{1, \dots, D\}$.

Proof. Let us first write out the matrices $\mathbf{M}^{(\ell)}$ and $\mathbf{B}^{(\ell)}$ elementwise. This will allow us to compare the coefficients we get later with those we get from the recursive definitions. From (3.89), we have

$$\begin{aligned}
\mathbf{M}_{ik}^{(\ell)} &= \bigoplus_{j=1}^{D} \mathbf{M}_{ij}^{(\ell-1)} \otimes \mathbf{G}_{jk}^{(\ell)} && (3.92)\\
&= \begin{cases} \mathbf{M}_{ik}^{(\ell-1)} \oplus \mathbf{M}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \otimes \mathbf{M}_{\ell k}^{(\ell-1)} & k \neq \ell \\ \mathbf{M}_{ik}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* & k = \ell \end{cases} && (3.93)\\
&= \begin{cases} \mathbf{M}_{ik}^{(\ell-1)} \oplus \mathbf{M}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \otimes \mathbf{M}_{\ell k}^{(\ell-1)} & k \neq \ell \\ \mathbf{M}_{ik}^{(\ell-1)} \otimes \left(\mathbf{1} \oplus \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \otimes \mathbf{M}_{\ell\ell}^{(\ell-1)}\right) & k = \ell \end{cases} && (3.94)\\
&= \begin{cases} \mathbf{M}_{ik}^{(\ell-1)} \oplus \mathbf{M}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \otimes \mathbf{M}_{\ell k}^{(\ell-1)} & k \neq \ell \\ \mathbf{M}_{ik}^{(\ell-1)} \otimes \oplus \mathbf{M}_{ik}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \otimes \mathbf{M}_{\ell\ell}^{(\ell-1)} & k = \ell \end{cases} && (3.95)\\
&= \mathbf{M}_{ik}^{(\ell-1)} \oplus \mathbf{M}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \otimes \mathbf{M}_{\ell k}^{(\ell-1)} && (3.96)
\end{aligned}$$

and, from (3.90)

$$\begin{aligned}
\mathbf{B}_{ik}^{(\ell)} &= \bigoplus_{j=1}^{D} \mathbf{B}_{ij}^{(\ell-1)} \otimes \left(\mathbf{G}^{(\ell)}\right)_{jk} && (3.97)\\
&= \begin{cases} \mathbf{B}_{ik}^{(\ell-1)} \oplus \mathbf{B}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \otimes \mathbf{M}_{\ell k}^{(\ell-1)} & k \neq \ell \\ \mathbf{B}_{ik}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* & k = \ell \end{cases} && (3.98)\\
&= \begin{cases} \mathbf{B}_{ik}^{(\ell-1)} \oplus \mathbf{B}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \otimes \mathbf{M}_{\ell k}^{(\ell-1)} & k \neq \ell \\ \mathbf{B}_{ik}^{(\ell-1)} \otimes \left(\mathbf{1} \oplus \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \otimes \mathbf{M}_{\ell\ell}^{(\ell-1)}\right) & k = \ell \end{cases} && (3.99)\\
&= \begin{cases} \mathbf{B}_{ik}^{(\ell-1)} \oplus \mathbf{B}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \otimes \mathbf{M}_{\ell k}^{(\ell-1)} & k \neq \ell \\ \mathbf{B}_{ik}^{(\ell-1)} \oplus \mathbf{B}_{ik}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \otimes \mathbf{M}_{\ell\ell}^{(\ell-1)} & k = \ell \end{cases} && (3.100)\\
&= \mathbf{B}_{ik}^{(\ell-1)} \oplus \mathbf{B}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \otimes \mathbf{M}_{\ell k}^{(\ell-1)} && (3.101)
\end{aligned}$$

The proof is by induction. The base case of $\ell = 1$ is proven in Prop. 3.6.5. Now, we assume by the inductive hypothesis that $\mathbf{Y} = \mathbf{Y}^{(\ell-1)} \otimes \mathbf{M}^{(\ell-1)} \otimes \mathbf{B}^{(\ell-1)}$ holds. The rest of the proof mirrors that of Prop. 3.6.5. On one hand, by a result in the Appendix, we have for any i

$$\mathbf{Y}_{i\ell} = \bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \mathbf{M}_{j\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \oplus \mathbf{B}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \qquad \text{(Lemma 3.6.8)} \qquad (3.102)$$

And, on the other by the inductive hypothesis, we have for any i and k

$$\mathbf{Y}_{ik} = \bigoplus_{j=1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \mathbf{M}_{j\ell}^{(\ell-1)} \oplus \mathbf{B}_{ik}^{(\ell-1)} \tag{3.103}$$

$$= \bigoplus_{j=\ell}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{j\ell}^{(\ell-1)} \oplus \mathbf{B}_{ik}^{(\ell-1)} \qquad \text{(definition of } \mathbf{Y}^{(\ell)}\text{)} \tag{3.104}$$

$$= (\mathbf{Y}_{i\ell}) \otimes \mathbf{M}_{\ell k}^{(\ell-1)} \oplus \bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{kj} \otimes \mathbf{M}_{jk}^{(\ell-1)} \oplus \mathbf{B}_{ik}^{(\ell-1)} \qquad \text{(pulling out the first term)} \tag{3.105}$$

$$\begin{aligned} = &\left(\bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \mathbf{M}_{j\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^{*} \oplus \mathbf{B}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^{*} \right) \otimes \mathbf{M}_{\ell k}^{(\ell-1)} \\ &\oplus \bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{kj} \otimes \mathbf{M}_{jk}^{(\ell-1)} \oplus \mathbf{B}_{ik}^{(\ell-1)} \qquad \text{(plugging in Eq. (3.102))} \end{aligned} \tag{3.106}$$

$$\begin{aligned} = &\bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \mathbf{M}_{j\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^{*} \otimes \mathbf{M}_{\ell k}^{(\ell-1)} \\ &\oplus \mathbf{B}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^{*} \otimes \mathbf{M}_{\ell k}^{(\ell-1)} \oplus \bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \mathbf{M}_{jk}^{(\ell-1)} \oplus \mathbf{B}_{ik}^{(\ell-1)} \qquad \text{(distributivity)} \end{aligned} \tag{3.107}$$

$$\begin{aligned} = &\bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \mathbf{M}_{j\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^{*} \otimes \mathbf{M}_{\ell k}^{(\ell-1)} \oplus \bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \mathbf{M}_{jk}^{(\ell-1)} \\ &\oplus \mathbf{B}_{ik}^{(\ell-1)} \oplus \mathbf{B}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^{*} \otimes \mathbf{M}_{\ell k}^{(\ell-1)} \qquad \text{(commutativity of } \oplus\text{)} \end{aligned} \tag{3.108}$$

$$\begin{aligned} = &\bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \left(\mathbf{M}_{j\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^{*} \otimes \mathbf{M}_{\ell k}^{(\ell-1)} \oplus \mathbf{M}_{jk}^{(\ell-1)}\right) \\ &\oplus \mathbf{B}_{ik}^{(\ell-1)} \oplus \mathbf{B}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^{*} \otimes \mathbf{M}_{\ell k}^{(\ell-1)} \qquad \text{(distributivity)} \end{aligned} \tag{3.109}$$

$$\begin{aligned} = &\bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \left(\mathbf{M}_{jk}^{(\ell-1)} \oplus \mathbf{M}_{j\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^{*} \otimes \mathbf{M}_{\ell k}^{(\ell-1)}\right) \\ &\oplus \left(\mathbf{B}_{ik}^{(\ell-1)} \oplus \mathbf{B}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^{*} \otimes \mathbf{M}_{\ell k}^{(\ell-1)}\right) \end{aligned} \tag{3.110}$$

Comparing the coefficients in Eq. (3.110) with those in Eqs. Eq. (3.96) and Eq. (3.101), we see that they match. This means that we can express the expression in terms of a matrix product:

$$\bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \left(\mathbf{M}_{jk}^{(\ell-1)} \oplus \mathbf{M}_{j\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^{*} \otimes \mathbf{M}_{\ell k}^{(\ell-1)}\right) \tag{3.111}$$

$$\oplus \left(\mathbf{B}_{ik}^{(\ell-1)} \oplus \mathbf{B}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^{*} \otimes \mathbf{M}_{\ell k}^{(\ell-1)}\right)$$

$$= \bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \mathbf{M}_{jk}^{(l)} \oplus \mathbf{B}_{ik}^{(l)} \tag{3.112}$$

$$= \bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell)} \otimes \mathbf{M}_{jk}^{(l)} \oplus \mathbf{B}_{ik}^{(l)} \tag{3.113}$$

$$= \left(\mathbf{Y}^{(\ell)} \otimes \mathbf{M}^{(l)}\right)_{ik} \oplus \mathbf{B}_{ik}^{(l)}, \tag{3.114}$$

meaning that $\mathbf{Y} = \mathbf{Y}^{(\ell)} \otimes \mathbf{M}^{(\ell)} \otimes \mathbf{B}^{(\ell)}$, which concludes the proof.

■

We can now combine the above results into our main theorem, which, besides providing the basis of the correctness of Lehmann's algorithm, also gives the intuition of what the subsequent elimination steps detailed above do.

Theorem 3.6.6. *Lehmann computes, for a given matrix $\mathbf{M}$ over a closed semiring, a solution to the equation*

$$\mathbf{Y} = \mathbf{Y} \otimes \mathbf{M} \oplus \mathbf{I}, \tag{3.115}$$

which takes the form

$$\mathbf{Y} = \bigotimes_{\ell=1}^{D} \mathbf{G}^{(\ell)}. \tag{3.116}$$

Proof. By Thm. 3.6.5, all intermediate operations satisfy the equation $\mathbf{Y} = \mathbf{Y}^{(n)} \otimes \mathbf{M}^{(n)} \oplus \mathbf{B}^{(n)}$. This means that for $n = D$, we have

$$\mathbf{Y} = \mathbf{Y}^{(D)} \otimes \mathbf{M}^{(D)} \oplus \mathbf{B}^{(D)} \tag{3.117}$$

$$= \mathbf{O} \otimes \mathbf{M}^{(D)} \oplus \mathbf{B}^{(D)} \qquad \text{(definition of } \mathbf{Y}^{(D)}\text{)} \tag{3.118}$$

$$= \mathbf{B}^{(D)} \qquad \text{(definition of zero)} \tag{3.119}$$

$$= \bigotimes_{\ell=1}^{D} \mathbf{G}^{(\ell)} \qquad \text{(definition of } \mathbf{B}^{(D)} \text{ when } \mathbf{B} = \mathbf{I}\text{)} \tag{3.120}$$

■

This leads us to the algorithm Alg. 11, which we will call the matrix version of Lehmann's algorithm. It directly applies the result from Thm. 3.6.6 to compute the Kleene closure.

This version of the algorithm, however, does not look like the one presente in Alg. 8. Furthermore, the D matrix multiplications hint that the runtime complexity of the algorithm may be quartic. However, we now complete the derivation by writing Alg. 11 in its more familiar form where the matrix multiplication Line 5 is expanded. This expansion also serves to demonstrate that the algorithm can be implemented in $\mathcal{O}\left(D^3\right)$ time, rather than $\mathcal{O}\left(D^4\right)$, due to the sparsity patterns of $\mathbf{G}^{(j)}$.

Algorithm 11 Lehmann's algorithm in matrix form

```
1. def Lehmann($\mathbf{M}$):
2.     ▷ $\mathbf{M}$ is a $D \times D$ matrix over a closed semiring
3.     $\mathbf{R}^{(0)} \leftarrow \mathbf{M}$
4.     for $j \leftarrow 1$ up to $D$ :
5.         $\mathbf{R}^{(j)} \leftarrow \mathbf{R}^{(j-1)} \otimes \mathbf{G}^{(j)}$
6.     ▷ $\mathbf{R}^{(D)} = \mathbf{M} \otimes \mathbf{M}^* = \mathbf{M}^+$
7.     return $\mathbf{I} \oplus \mathbf{R}^{(D)}$
```

Proposition 3.6.6. *The following equation holds*

$$\left(\mathbf{R}^{(j-1)} \otimes \mathbf{G}^{(j)}\right)_{ik} = \mathbf{R}^{(j-1)}_{ik} \oplus \mathbf{R}^{(j-1)}_{ij} \otimes \left(\mathbf{R}^{(j-1)}_{jj}\right)^* \otimes \mathbf{R}^{(j-1)}_{jk} \tag{3.121}$$

Proof. We consider two cases.

Case 1: $i = k$**.**

$$\left(\mathbf{R}^{(j-1)} \otimes \mathbf{G}^{(j)}\right)_{kk} = \bigoplus_{d=1}^{D} \mathbf{R}^{(j-1)}_{kd} \otimes \mathbf{G}^{(j)}_{dk} \qquad \text{(matrix multiplication)} \tag{3.122}$$

$$= \mathbf{R}^{(j-1)}_{kk} \otimes \mathbf{G}^{(j)}_{kk} \qquad \text{(sparsity pattern)} \tag{3.123}$$

$$= \mathbf{R}^{(j-1)}_{kk} \otimes \left(\mathbf{R}^{(j-1)}_{kk}\right)^* \qquad \text{(definition of } \mathbf{G}^{(j)}_{jk}\text{)} \tag{3.124}$$

$$= \mathbf{R}^{(j-1)}_{kk} \oplus \mathbf{R}^{(j-1)}_{kj} \otimes \left(\mathbf{R}^{(j-1)}_{jj}\right)^* \otimes \mathbf{R}^{(j-1)}_{jk} \qquad \text{(Prop. 3.6.9)} \tag{3.125}$$

Case 2: $i \neq k$**.**

$$\left(\mathbf{R}^{(j-1)} \otimes \mathbf{G}^{(j)}\right)_{ik} = \bigoplus_{d=1}^{D} \mathbf{R}^{(j-1)}_{id} \otimes \mathbf{G}^{(j)}_{dk} \qquad \text{(matrix multiplication)} \tag{3.126}$$

$$= \mathbf{R}^{(j-1)}_{ik} \otimes \left(\mathbf{1} \oplus \mathbf{G}^{(j)}_{jk}\right) \qquad \text{(sparsity pattern)} \tag{3.127}$$

$$= \mathbf{R}^{(j-1)}_{ik} \oplus \mathbf{R}^{(j-1)}_{ij} \otimes \mathbf{G}^{(j)}_{jk} \qquad \text{(distributivity)} \tag{3.128}$$

$$= \mathbf{R}^{(j-1)}_{ik} \oplus \mathbf{R}^{(j-1)}_{ij} \otimes \left(\mathbf{R}^{(j-1)}_{jj}\right)^* \otimes \mathbf{R}^{(j-1)}_{jk} \qquad \text{(definition of } \mathbf{G}^{(j)}_{jk}\text{)} \tag{3.129}$$

■

Now, to derive Lehmann's algorithm as Lehmann (1977) himself presented it we write out the matrix $\mathbf{R}^{(j)}$ on Line 5 of Alg. 11 using Prop. 3.6.6. This manipulation yields Alg. 12, which is line for line equivalent to Lehmann (1977).

Notice that Alg. 12 uses D different matrices during execution, which is very inefficient. Since the iterate in iteration j only depends on the results from the iteration $j - 1$, it is easy to see how the same algorithm can be implemented using only two matrices. This is left as an exercise.

Finishing up the Proof of Correctness.

Algorithm 12 Lehmann's algorithm

```
def Lehmann($\mathbf{M}$):
  ▷ $\mathbf{M}$ is a $D \times D$ matrix over a closed semiring
  $\mathbf{R}^{(0)} \leftarrow \mathbf{M}$
  for $j \leftarrow 1$ up to $D$ :
    for $i \leftarrow 1$ up to $D$ :
      for $k \leftarrow 1$ up to $D$ :
        $\mathbf{R}_{ik}^{(j)} \leftarrow \mathbf{R}_{ik}^{(j-1)} \oplus \mathbf{R}_{ij}^{(j-1)} \otimes \left(\mathbf{R}_{jj}^{(j-1)}\right)^{*} \otimes \mathbf{R}_{jk}^{(j-1)}$
  return $\mathbf{I} \oplus \mathbf{R}^{(D)}$
```

Semirings which are not fields. If you have followed the derivation so far carefully, you might have noticed that we have only shown that Lehmann only computes *a* solution to Eq. (3.76). We have said nothing about how this relates to the infinite sum of the Kleene closure or indeed anything about the second axiomatic property $\mathbf{M}^* = \mathbf{I} \oplus \mathbf{M}^* \otimes \mathbf{M}$.

This section brings these considerations together and specifies precisely where the results of the derivation are applicable. Since the formal treatment of the results is again rather involved and would require multiple additional pages, we only list the relevant results and discuss their context. Some of the easier proofs are left as an exercise.

To do so formally, we first define another important algebraic structure, that of a **dioid**. For that, however, we have to introduce the notion of a canonical ordering of a semiring.

Let us recall **(partial) orders** first, which generalize the concept of an ordering.

Definition 3.6.15 (Partial Order)**.** *A **partially ordered set**, or **poset**, is the tuple* $(\mathbb{K}, \preccurlyeq)$ *of a set together with a binary relation* $\preccurlyeq$ *indicating that, for* certain pairs of elements *in the set, one of the elements precedes the other in the ordering. The relation* $\preccurlyeq$ *is called a partial order and must satisfy 3 axioms:* $\forall a, b, c \in \mathbb{K}$

(i) ***Reflexivity****:* $a \preccurlyeq a$

(ii) ***Antisymmetry****: If* $a \preccurlyeq b$ *and* $b \preccurlyeq a$ *then* $a = b$

(iii) ***Transitivity****: If* $a \preccurlyeq b$ *and* $b \preccurlyeq c$ *then* $a \preccurlyeq c$

The generalization of the regular order comes from the fact that not every pair of elements needs to be comparable. That is, there may be pairs of elements for which neither element precedes the other. The special case of partial orders in which *every pair of nodes is comparable* is called a **linear** or **total order**.

It turns out there is a general way to define partial orders on some classes of semirings.

Definition 3.6.16 (Dioid)**.** *Let* $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ *be a semiring. We define the relation* $\preccurlyeq$ *as*

$$a \preccurlyeq b \iff \exists x \in \mathbb{K} : \; a \oplus x = b. \tag{3.130}$$

We say the semiring $\mathcal{W}$ *is a **dioid** if* $\preccurlyeq$ *is a valid partial order. Then,* $\preccurlyeq$ *is called the **canonical** or **natural order** over* $\mathbb{K}$*.*

We will show that we can define the **minimal** solution $\mathbf{Y}_{\min}$ such that

$$\mathbf{Y}_{\min} \preccurlyeq \mathbf{Y} \quad \forall\, \mathbf{Y} \text{ such that } \mathbf{Y} = \mathbf{Y} \otimes \mathbf{M} \oplus \mathbf{B}, \tag{3.131}$$

i.e., there exists a solution to Eq. (3.76) which is comparable to all other solutions and is in fact the minimal one.

It turns out that the minimal solution is exactly the Kleene closure $\mathbf{M}^*$ of $\mathbf{M}$ as long as the infinite sum is finite:

Proposition 3.6.7. *If the semiring $\mathcal{W}$ is a dioid and there exists $K \in \mathbb{N}$ such that $\mathbf{M}^* = \mathbf{M}^{(K)}$, then $\mathbf{Y} = \mathbf{B} \otimes \mathbf{M}^*$ is the* minimal solution *to the system $\mathbf{Y} = \mathbf{Y} \otimes \mathbf{M} \otimes \mathbf{B}$ and $\mathbf{Y} = \mathbf{M}^* \otimes \mathbf{B}$ is the* minimal solution *to the system $\mathbf{Y} = \mathbf{M} \otimes \mathbf{Y} \otimes \mathbf{B}$.*

Proof. Let $\mathbf{Y}$ be an arbitrary solution to Eq. (3.76). Then we can write

$$\mathbf{Y} = \mathbf{Y} \otimes \mathbf{M} \oplus \mathbf{B} \tag{3.132}$$

$$= (\mathbf{Y} \otimes \mathbf{M} \oplus \mathbf{B}) \otimes \mathbf{M} \oplus \mathbf{B} \tag{3.133}$$

$$= \mathbf{Y} \otimes \mathbf{M}^2 \oplus \mathbf{B} \otimes (\mathbf{I} \oplus \mathbf{M}) . \tag{3.134}$$

In general, this means that for any $n \in \mathbb{N}$, we have

$$\mathbf{Y} = \mathbf{Y} \otimes \mathbf{M}^n \oplus \mathbf{B} \otimes (\mathbf{I} \oplus \mathbf{M} \oplus \ldots \oplus \mathbf{M}^n) . \tag{3.135}$$

Therefore, for any $n \geq K$, we get

$$\mathbf{Y} = \mathbf{Y} \otimes \mathbf{M}^n \oplus \mathbf{B} \otimes \mathbf{M}^*, \tag{3.136}$$

meaning that $B \otimes \mathbf{M}^*$ is "smaller" than any solution $\mathbf{Y}$ of Eq. (3.76), and is thus the minimal solution to Eq. (3.76).

■

Furthermore, it can be shown that the solution computed by Lehmann is *also minimal*:

Proposition 3.6.8. *If the semiring $\mathcal{W}$ is a dioid, then $\mathbf{Y}$ computed by Lehmann is the* minimal solution *to the system $\mathbf{Y} = \mathbf{Y} \otimes \mathbf{M} \otimes \mathbf{B}$.*[16]

Since *both solutions are minimal* and we have a total order over them, they are, due to antisymmetry, identical. Furthermore, since $\mathbf{M}$ and $\mathbf{M}^*$ commute if $\mathbf{M}^* = \mathbf{M}^{(K)}$ for some $K \in \mathbb{N}$, we can conclude that for the special case of $\mathbf{B} = \mathbf{I}$, the solution computed by Lehmann satisfies both axiomatic properties. This mean that Lehmann computes the Kleene closure of the matrix! This concludes our proof of correctness of Lehmann for semirings which are not fields.

Commutative Fields. Lehmann also correctly computes the Kleene closure of a matrix $\mathbf{M}$ over any commutative field, whenever $\mathbf{M}^*$ exists. As mentioned in the section on GaussJordan, the in the case that the infinite sum converges, it equals $(\mathbf{I} - \mathbf{M})^{-1}$. Note that because we assume commutativity (as in the case of the reals, for example), we only have to consider one of the axiomatic properties, since they are equivalent. We thus have

$$\mathbf{M}^* = \bigoplus_{n=0}^{\infty} \mathbf{M}^n = (\mathbf{I} - \mathbf{M})^{-1}, \tag{3.137}$$

which is the (unique) solution of the equation $\mathbf{Y} = \mathbf{I} \oplus \mathbf{Y} \otimes \mathbf{M}$ in the real field. We saw above that Lehmann computes a valid solution to this equation, and the uniqueness means that Lehmann returns the correct result for any matrix $\mathbf{M}$ for which the infinite sum in Eq. (3.41) exists.

[16] To be fully precise, there is another very technical condition, namely that the semiring has to be a **topological** dioid. We can assume this holds for the dioids we will encounter in the course.

In the case of reals, however, we can be even more general. We established the correctness of Lehmann whenever the Kleene closure of the matrix is defined. Concretely, this would hold in the case when for all elements of $\mathbf{M}$ it holds that $|\mathbf{M}_{ij}| < \frac{1}{D}$, meaning that $\mathbf{M}$ is sufficiently close to the zero matrix. However, the elements of $(\mathbf{I} - \mathbf{M})^{-1}$ are just *rational functions in the elements of* $\mathbf{M}$, and Lehmann computes rational functions of its inputs as well. Therefore, we have two rational functions which coincide in an open subset of the input space (the neighborhood of the zero matrix). This means that they must coincide everywhere in their common domain, implying that we can generalize the procedure to any matrix $\mathbf{M}$ such that $(\mathbf{I} - \mathbf{M})^{-1}$ exists. If we then define

$$x^* = \begin{cases} \frac{1}{1-x} & x \neq 1 \\ \text{undefined} & x = 1 \end{cases} \tag{3.138}$$

for any $x \in \mathbb{R}$ no matter the convergence of the infinite series, Lehmann computes $(\mathbf{I} - \mathbf{M})^{-1}$ almost always.

Appendix to Lehmann's Algorithm

Proposition 3.6.9.

$$\mathbf{M} \otimes \mathbf{M}^* = \mathbf{M} \oplus \mathbf{M} \otimes \mathbf{M}^* \otimes \mathbf{M} \tag{3.139}$$

Proof.

$$\begin{align} \mathbf{M} \otimes \mathbf{M}^* &= \mathbf{M} \otimes (\mathbf{I} \oplus \mathbf{M}^* \otimes \mathbf{M}) && \text{(Kleene axiom)} \tag{3.140} \\ &= \mathbf{M} \oplus \mathbf{M} \otimes \mathbf{M}^* \otimes \mathbf{M} \tag{3.141} \end{align}$$

■

Lemma 3.6.6. *Let $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1}, ^*)$ be a closed semiring. For any $a \in \mathbb{K}$, $b \otimes a^*$ is a solution to the equation*

$$y = y \otimes a \oplus b \tag{3.142}$$

and $a^* \otimes b$ is a solution to the equation

$$y = a \otimes y \oplus b. \tag{3.143}$$

Proof. We just prove it for the first case as the second one is analogous.

Let $y = b \otimes a^*$. Then the right hand side of Eq. (3.142) equals

$$b \otimes a^* \otimes a \oplus b = b \otimes (a^* \otimes a \oplus \mathbf{1}) = b \otimes a^*$$

■

Lemma 3.6.7. *A solution to*

$$\mathbf{Y}_{i1} = \mathbf{Y}_{i1} \otimes \mathbf{M}_{11} \oplus \bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{j1} \oplus \mathbf{B}_{i1} \tag{3.144}$$

takes the form

$$\mathbf{Y}_{i1} = \bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{j1} \otimes \mathbf{M}_{11}^* \oplus B_{i1} \otimes \mathbf{M}_{11}^*. \tag{3.145}$$

Proof. The expression Eq. (3.144) is of the form Eq. (3.142) for $a = \mathbf{M}_{11}$ and $b = \bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{j1} \oplus \mathbf{B}_{i1}$. Therefore, by Lemma 3.6.6, a solution to it is

$$\mathbf{Y}_{i1} = b \otimes a^* = \left(\bigoplus_{j=2}^{D} \mathbf{Y}_{ij} \otimes \mathbf{M}_{j1} \oplus \mathbf{B}_{i1} \right) \otimes \mathbf{M}_{i1}^* = \bigoplus_{j=2}^{D} \mathbf{Y}_{1j} \otimes \mathbf{M}_{j1} \otimes \mathbf{M}_{11}^* \oplus \mathbf{B}_{i1} \otimes \mathbf{M}_{11}^*. \quad (3.146)$$

■

Lemma 3.6.8. *For any $\ell \in \{1, \ldots, D\}$, a solution to*

$$\mathbf{Y}_{i\ell} = \mathbf{Y}_{i\ell}^{(\ell-1)} \otimes \mathbf{M}_{\ell\ell}^{(\ell-1)} \oplus \bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \mathbf{M}_{jl}^{(\ell-1)} \oplus \mathbf{B}_{il}^{(\ell-1)} \quad (3.147)$$

takes the form

$$\mathbf{Y}_{i\ell} = \bigoplus_{j=\ell+1}^{D} \mathbf{Y}_{ij}^{(\ell-1)} \otimes \mathbf{M}_{j\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^* \oplus \mathbf{B}_{i\ell}^{(\ell-1)} \otimes \left(\mathbf{M}_{\ell\ell}^{(\ell-1)}\right)^*. \quad (3.148)$$

Proof. The proof follows the same steps as proof for Lemma 3.6.8 by taking into account the definition of $\mathbf{Y}^{(\ell-1)}$ with the first $\ell - 1$ columns being 0. ■

3.6.7 Strongly Connected Components

We can often consider the degree of connectivity within a graph as a measure of its complexity—the more densely the graph is connected, the more complex the WFSA/system it is describing is. In that case, sparse graphs (graphs in which not many nodes are connected directly) are especially "simple". We can sometimes exploit sparsity of graphs to make algorithms such as those for computing the pathsum (substantially) more efficient.

This subsection introduces the notions of strong connectivity and strongly connected components, together with algorithms for computing them. We will see that the complexity of the algorithms we looked at in this chapter so far depends heavily on these concepts, and we will present a simple extension of Lehmann which can make it critically more efficient.

Definition 3.6.17 (Strongly Connected Components)**.** *A directed graph is said to be* ***strongly connected*** *if every node is reachable from every other node, i.e., there is a path* in each direction *between each pair of nodes of the graph. In a directed graph that may not itself be strongly connected, a pair of nodes are strongly connected to each other if there is a path in each direction between them. The* ***strongly connected components (SCCs)*** *of a directed graph form a partition into subgraphs that are themselves strongly connected, and are maximal with this property, meaning that, for each strongly connected component $\mathcal{S}$ of a graph $\mathcal{G}$, there is no other strongly connected subgraph in $\mathcal{G}$ which would contain $\mathcal{S}$ as a subgraph and would itself be strongly connected.*

The fact that SCCs form a **partition** of the original graph means that any node in the original graph is contained in exactly one strongly connected component.

We can look at the strong connectivity of pairs of nodes as a binary relation. It is in fact an **equivalence relation**, and the SCCs correspond to its **equivalence classes**.

We saw that Lehmann has a very tight connection to solving linear systems through the GaussJordan algorithm. Strongly connected components arise often in solving linear systems in which the individual equations only depend on a small subset of variables, but the number of variables and equations all together is large (in the millions). The naïve application of the cubic-time Lehmann would result in prohibitive runtime, but by exploiting the sparsity of the system, we will see that they can be solved quite efficiently.

A useful operation on the SCCs is **concentration**, which we define next.

Definition 3.6.18. *Let $\mathcal{G}$ be a graph. The **concentration** of a SCC $\mathcal{S}$ is an operation which constructs a single node $q_{\mathcal{S}}$ from the entire SCC. We construct the **condensation** $\mathcal{G}'$ of $\mathcal{G}$ by concentrating all SCCs in $\mathcal{G}$ and putting a directed edge from $q_{\mathcal{S}}$ to $q_{\mathcal{S}'}$ if there is a directed edge going from $\mathcal{S}$ into $\mathcal{S}'$ in $\mathcal{G}$,*

The condensation is a DAG—if there were a directed cycle in $\mathcal{G}'$, all the nodes in the SCCs withing the cycle would be strongly connected. In fact, a directed graph is acyclic *if and only if* it has no strongly connected subgraphs with more than one vertex, because a directed cycle is strongly connected and every nontrivial strongly connected component contains at least one directed cycle.

There are in fact multiple known algorithms to both test the strong connectivity of a graph and find all its connected components. We focus on one of the simpler ones, **Kosaraju's algorithm**.

Recall the depth-first search from Alg. 4. We will denote the finishing time of the node q with $f[q]$. Before diving into Kosaraju's algorithm, we can make an important remark.

Proposition 3.6.10. *If the finishing time of a node q is larger than the finishing time of node q', one of the three options can hold:*

(i) q is an ancestor of q' and q' is not *an ancestor of q*

(ii) q is an ancestor of q' and q' is an ancestor of q' (they belong to the same SCC)

(iii) Neither of the two nodes is an ancestor of the other.

Crucially, it can not *hold that q' is an ancestor of q and q is* not *an ancestor of q'.*

Proof. To see that, suppose that this fourth option holds. If we visit q' *before* q in DepthFirstSearch, then $f[q'] > f[q]$, since we first have to process q'. On the other hand, if we visit q before q' (for example, by starting DepthFirstSearch in q), then we will again finish the expansion of q before we get to q' by the assumption that q is not an ancestor of q'. This leads to a contradiction, meaning that the fourth option indeed can not happen. ■

With this consideration, we can take a closer look at Kosaraju's algorithm. Its main idea is that if we traverse the graph in the order of increasing finishing times, and we observe a "forward connection", i.e., an edge between from a node q' to a node q such that $f[q] > f[q']$, then we know, from the Prop. 3.6.10, that these two nodes belong to the same SCC. On the other hand, for any node q' from which we can not reach q, it, obviously, holds that it will not belong to the same SCC as q. The entire algorithm is outlined in Alg. 13.

The algorithm defines the SCCs in terms of their **root** nodes, i.e., the node of the SCC which was visited first (and has the largest finishing time within that SCC). The root nodes are traversed in increasing finishing time (line 4). Once the root q is selected such that it has not been visited yet, its SCC is initialized, at the start with q as the only element. Then, all *ancestor* nodes q' of q are traversed. Again, Prop. 3.6.10 implies that this implies that q' belong to the SCC whose root is q. Lines 7–9 traverse *all* ancestors of q with the queue of encountered ancestors, meaning that if a node q' belongs to the same SCC as q, it will be traversed and, thus, put into the correct SCC. If, on the other hand, a node is *not* put on the queue `queue` at any point in lines 7–9, it does not belong to the SCC being constructed.

Algorithm 13 The Kosaraju algorithm

```
def Kosaraju($\mathcal{G}$):
  SCCs ← []                                  ▷ Initialize an empty list for storing the SCCs.
  visited ← {}                               ▷ Initialize an empty set for storing processed nodes to avoid loops
  for $q \in$ topo ($\mathcal{G}$) \ visited :
    ▷ Nodes are traversed in increasing finishing time. $q$ will be the root of the new SCC.
    $\mathcal{S} \leftarrow \{q\}$           ▷ A set containing the nodes of the SCC in construction.
    queue ← [$q$]                            ▷ A queue containing the reachable nodes in the SCC yet to be processed.
    while |queue| > 0 :
      pop $q'$ from queue
      add $q'$ to $\mathcal{S}$ and visited
      for every transition $q'' \xrightarrow{\bullet} q'$ in $\mathcal{G}$ :
        if $q'' \notin$ visited :            ▷ For any node $q''$ added to queue, we can reach the root $q$ from $q''$
          add $q''$ to queue
    add $c$ to SCCs
  return SCCs
```

Complexity The algorithm traverses the graph twice: once for calculating the finishing times of the nodes and once for constructing the components. Therefore, it runs in time $\Theta(V + E)$, where V and E denote the number of nodes and edges in $\mathcal{G}$, respectively. Since all edges have to be considered to determine the SCCs, this is asymptotically optimal, however, there exist algorithms that do not have to traverse the graph twice, such as **Tarjan's algorithm**, and are therefore faster.

3.7 Acceptance by Weighted Finite-state Automata

Testing the acceptance of a string $\boldsymbol{y} \in \Sigma^*$ with a WFSA $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ is one of the most basic operations we might want to do with $\mathcal{A}$. In contrast to the unweighted case, where we only talk about accepting a string or not accepting it, the weighted version consider the *weight* with which $\boldsymbol{y}$ is accepted, i.e., $\mathcal{A}(\boldsymbol{y})$. Recall that we refer to $\mathcal{A}(\boldsymbol{y})$ as the *string weight*. The accepting weight can also be extended to a *set* of strings S:

$$\mathcal{A}(S) \overset{\text{def}}{=} \bigoplus_{\boldsymbol{y} \in S} \mathcal{A}(\boldsymbol{y}). \tag{3.149}$$

We refer to $\mathcal{A}(S)$ as the **string set weight**.

We first look at the case of the weighted acceptance of a single string and then trivially generalize it to sets. The principled way to do this is to construct, from $\mathcal{A}$, a new automaton, call it $\mathcal{A}_{\boldsymbol{y}}$, which only retains paths in $\mathcal{A}$ with the yield $\boldsymbol{y}$. Then, we can simply take the usual pathsum over $\mathcal{A}_{\boldsymbol{y}}$ to get the weight of the string $\boldsymbol{y}$ under $\mathcal{A}$.

The question now is, of course, how to obtain $\mathcal{A}_{\boldsymbol{y}}$. We do that in two steps:

(i) We construct a second automaton $\mathcal{A}^{(\boldsymbol{y})}$, which accepts *only* the string $\boldsymbol{y}$, which has the weight $\mathbf{1}$. In the simplest case $\mathcal{A}^{(\boldsymbol{y})}$ would take a form of a linear chain with 1 starting state and 1 final state, with the path $\boldsymbol{\pi}$ between them yielding $\boldsymbol{y}$.

(ii) We take the weighted intersection of the automata $\mathcal{A}$ and $\mathcal{A}^{(\boldsymbol{y})}$. By the definition of the intersection, the resulting automaton accepts the strings accepted by both automata, i.e., no other than $\boldsymbol{y}$. The weight of $\boldsymbol{y}$ is then $\mathcal{A}_{\boldsymbol{y}}(\boldsymbol{y}) \overset{\text{def}}{=} \mathcal{A}(\boldsymbol{y}) \otimes \mathcal{A}^{(\boldsymbol{y})}(\boldsymbol{y}) = \mathcal{A}(\boldsymbol{y}) \otimes \mathbf{1} = \mathcal{A}(\boldsymbol{y})$.

The full algorithm is presented in Alg. 14.

Algorithm 14 The Acceptance algorithm.

```
def Acceptance($\mathcal{A}, \boldsymbol{y}$):
    ▷ Construct $\mathcal{A}^{(\boldsymbol{y})}$ such that it accepts only $\boldsymbol{y}$ with weight $\mathbf{1}$.
    $\mathcal{A}^{(\boldsymbol{y})} \leftarrow (\Sigma, Q, \delta, \lambda, \rho)^{(\boldsymbol{y})}$ a WFSA over the same semiring
    add state $q_0$ to $\mathcal{A}^{(\boldsymbol{y})}$
    $\lambda^{(\boldsymbol{y})}(q_0) \leftarrow \mathbf{1}$
    for $a_n \in a_1 \cdots a_N$ :                                ▷Iterate through the string left to right
        add a new state $q_n$ to $\mathcal{A}^{(\boldsymbol{y})}$
        add the edge $q_{n-1} \xrightarrow{a_n/\mathbf{1}} q_n$ to $\mathcal{A}^{(\boldsymbol{y})}$
    $\rho^{(\boldsymbol{y})}(q_N) \leftarrow \mathbf{1}$
    $\mathcal{A}_{\boldsymbol{y}} \leftarrow$ Intersect $(\mathcal{A}, \mathcal{A}^{\boldsymbol{y}})$
    return $Z(\mathcal{A}_{\boldsymbol{y}})$                  ▷Compute the pathsum in linear time; $\mathcal{A}_{\boldsymbol{y}}$ is acyclic
```

The way to generalize the above procedure to testing the acceptance of *multiple* strings is simple. We construct $\mathcal{A}^{(S)}$ as the *union* of the automata $\mathcal{A}^{(\boldsymbol{y})}$ for $\boldsymbol{y} \in S$ and intersect $\mathcal{A}$ with $\mathcal{A}^{(S)}$. Then, instead of taking the complete pathsum of $\mathcal{A}(S)$, the weights of the individual strings $\boldsymbol{y} \in S$ are the pathsums between the initial states of $\mathcal{A}^{(S)}$ (corresponding to individual strings $\boldsymbol{y} \in S$) and the matching final states of the same strings. We again prove the correctness of the algorithm on the simple case of a single string $\boldsymbol{y}$.

Theorem 3.7.1. *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA and $\boldsymbol{y} \in \Sigma^*$. Acceptance computes $\mathcal{A}(\boldsymbol{y})$.*

Proof. The result follows simply

$$
\begin{aligned}
Z\left(\mathcal{A}_{\boldsymbol{y}}\right) &\overset{\text{def}}{=} \bigoplus_{\boldsymbol{\pi}\in\Pi(\mathcal{A}_{\boldsymbol{y}})} \lambda_{\boldsymbol{y}}\left(p\left(\boldsymbol{\pi}\right)\right)\otimes w_{\boldsymbol{y}}\left(\boldsymbol{\pi}\right)\otimes\rho_{\boldsymbol{y}}\left(n\left(\boldsymbol{\pi}\right)\right) && \text{(pathsum)} \quad (3.150)\\
&= \bigoplus_{\boldsymbol{\pi}\in\Pi(\mathcal{A})} \mathbb{1}\left\{\mathrm{s}\left(\boldsymbol{\pi}\right)=\boldsymbol{y}\right\}\otimes\lambda\left(p\left(\boldsymbol{\pi}\right)\right)\otimes w\left(\boldsymbol{\pi}\right)\otimes\rho\left(n\left(\boldsymbol{\pi}\right)\right) && \text{(intersection)} \quad (3.151)\\
&= \bigoplus_{\substack{\boldsymbol{\pi}\in\Pi(\mathcal{A}),\\ \mathrm{s}(\boldsymbol{\pi})=\boldsymbol{y}}} \lambda\left(p\left(\boldsymbol{\pi}\right)\right)\otimes w\left(\boldsymbol{\pi}\right)\otimes\rho\left(n\left(\boldsymbol{\pi}\right)\right) && \text{(change of notation)} \quad (3.152)\\
&= \mathcal{A}\left(\boldsymbol{y}\right) && \text{(definition of string sum)} \quad (3.153)
\end{aligned}
$$

■

$\mathcal{NP}$-completeness of the Most Likely String problem

In the previous paragraphs we proved that the problem of computing the pathsum in an acyclic WFSA can be solved in polynomial time leveraging the backward algorithm. In the probability semiring $\mathcal{W} = \langle [0,1], +, \cdot, 0, 1\rangle$, we can hence find the most likely path of the WFSA, and the corresponding most likely output, in linear time. However, the problem becomes much harder if we want to find the **Most Likely String of length l in a WFSA**, since the same output string could be output by several different paths. In fact, it can be shown that finding whether the most likely string of a given length n output by a WFSA has probability at least p is $\mathcal{NP}$-complete.

We will show the $\mathcal{NP}$-completeness of the Most Likely String problem by a reduction of the $\mathcal{NP}$-complete problem 3-Sat. The 3-Sat problem consists in showing whether or not there is a way to satisfy a boolean expression of the form $F_1 \wedge F_2 \wedge \cdots \wedge F_n$ where F_i is of the form $(\alpha_{i1} \vee \alpha_{i2} \vee \alpha_{i3})$. Here, the α_{ij} represent literals, either x_k or $\neg x_k$. On example of formula whose value is tested in a 3-Sat problem could be

$$(\neg x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_4 \vee x_3 \vee x_1). \quad (3.154)$$

The most obvious way to pursue this reduction is to create an automaton whose output corresponds to satisfying assignments of the clauses, e.g. a string of T's and F's, one letter for each variable, denoting whether that variable is true or false. There will be one section of the network for each clause, and the probabilities will be assigned in such a way that for the output probability to sum sufficiently high, for every clause the maximal string must have a path through the corresponding subnetwork. The problem with this technique is that such a string may be output by multiple paths through the subnetworks corresponding to some of the clauses, and zero paths through the parts of the network leading to other clauses. What we must do is to ensure that there is exactly one path through each clause's subnetwork. In order to do this, we augment the output with an initial sequence of the digits 1, 2, and 3. Each digit specifies which variable (the first, second, or third) of each clause satisfied that clause. This strategy allows construction of subnetworks for each clause with at most one path. Then, if there is a string whose probability indicates that it has n paths through the automaton, we know that the string satisfies every clause, and therefore satisfies the formula. Any string that does not correspond to a satisfying assignment will have fewer than n paths, and will not have a sufficiently high probability.

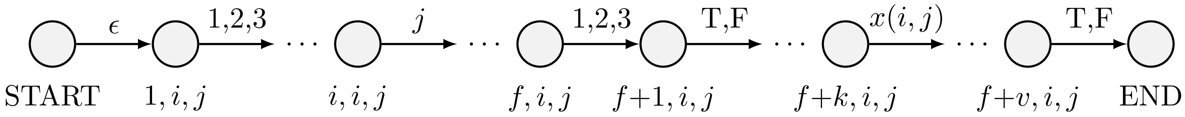
Theorem 3.7.2. *The Most Likely String problem is $\mathcal{NP}$-complete.*

Proof. Showing that the Most Likely String problem is $\mathcal{NP}$ is trivial. In fact, we can guess a most likely string of the given length n, compute its probability using polynomial-time dynamic programming techniques (e.g. the backward algorithm presented before), and finally verify that this probability is at least p.

To show $\mathcal{NP}$-completeness, we show how to reduce a 3-Sat formula to a WFSA Most Likely String problem. Let f represent the number of disjunctions and v represent the number of variables in the formula to satisfy. We will define a WFSA with $3f(f+v)+2$ states as follows:

- $3f(f+v)$ states (h,i,j), where $h \in [1, f+v]$ is an index for the nodes within a single path (e.g. $f+v = 3+4$ in Eq. (3.154)), $i \in [1, f]$ is and index for the number of boolean disjunctions in the original formula (e.g. $f = 3$ in Eq. (3.154)), and $j \in [1,3]$ is the index for the single literal in the selected disjunction (e.g. $j = 1$ indexes the first literal α_{i1} in the disjunction i);
- an initial state START, with an equiprobable epsilon transition to the all the $3f$ states of the form $(1,i,j)$;
- an accepting state END.

The basic form of the FSA is



where $x(i,j)$ is T if $\alpha_{i,j}$ is of the form x_k, and F if $\alpha_{i,j}$ is of the form $\neg x_k$. The arcs labeled with $(1,2,3)$ indicates a state transition with equal probabilities of emitting the symbols 1, 2, or 3. Similarly, the arcs labeled with (T,F) indicates a state transition with equal probabilities of emitting T or F. The WFSA corresponding to the 3-Sat in Eq. (3.154) is shown in Fig. 3.18.

We now show that if the given 3-Sat formula is satisfiable, then the constructed WFSA has a string with probability $\frac{1}{3f} \times \frac{1}{2^{v-1}}$, whereas if it is not satisfiable, the most likely string will have a lower probability. The proof is as follows. First, assume that there is some way to satisfy the 3-Sat formula. Let X_i denote T if the variable x_i takes on the value true, and let it denote F otherwise, under some assignment of values to variables that satisfies the formula. Also, let J_i denote the number 1, 2, or 3 respectively depending on whether for the i-th disjuction, α_{i1}, α_{i2}, or α_{i3} respectively satisfies formula i. If J_i could take on more than one value by this definition (that is, if F_i is satisfied in more than one way) then arbitrarily, we assign it to the lowest of these three. Since we are assuming here that the formula is satisfied under the assignment of values to variables, at least one of α_{i1}, α_{i2}, or α_{i3} must satisfy each clause. Then, the string $J_1 J_2 \ldots J_f X_1 X_2 \ldots X_v$ will be a most probable string.

There may be other most probable strings, but all will have the same probability and satisfy the formula. In the example given by Eq. (3.154), the string `131FTFT` would be the most likely string output by the WFSA. This string will have probability $\frac{1}{3f} \times \frac{1}{2^{v-1}}$ because there will be routes emitting that string through exactly f of the $3f$ subnetworks, and each route will have probability $\frac{1}{f} \times \frac{1}{3^f} \times \frac{1}{2^{v-1}}$.

On the other hand, assume that the formula is not satisfiable. Now, the most likely string will have a lower probability. Obviously, the string could not have higher probability, since the best path through any subnetwork has probability $\frac{1}{3^{f-1}} \times \frac{1}{2^{v-1}}$ and it is not possible to have routes through more than f of the $3f$ subnetworks. Therefore, the probability can be at most $\frac{1}{3f} \times \frac{1}{2^{v-1}}$. However, it cannot be this high, since if it were, then the string would have a path through f of the $3f$ networks, and would therefore correspond to a satisfaction of the formula. Thus, its probability must be lower. ■

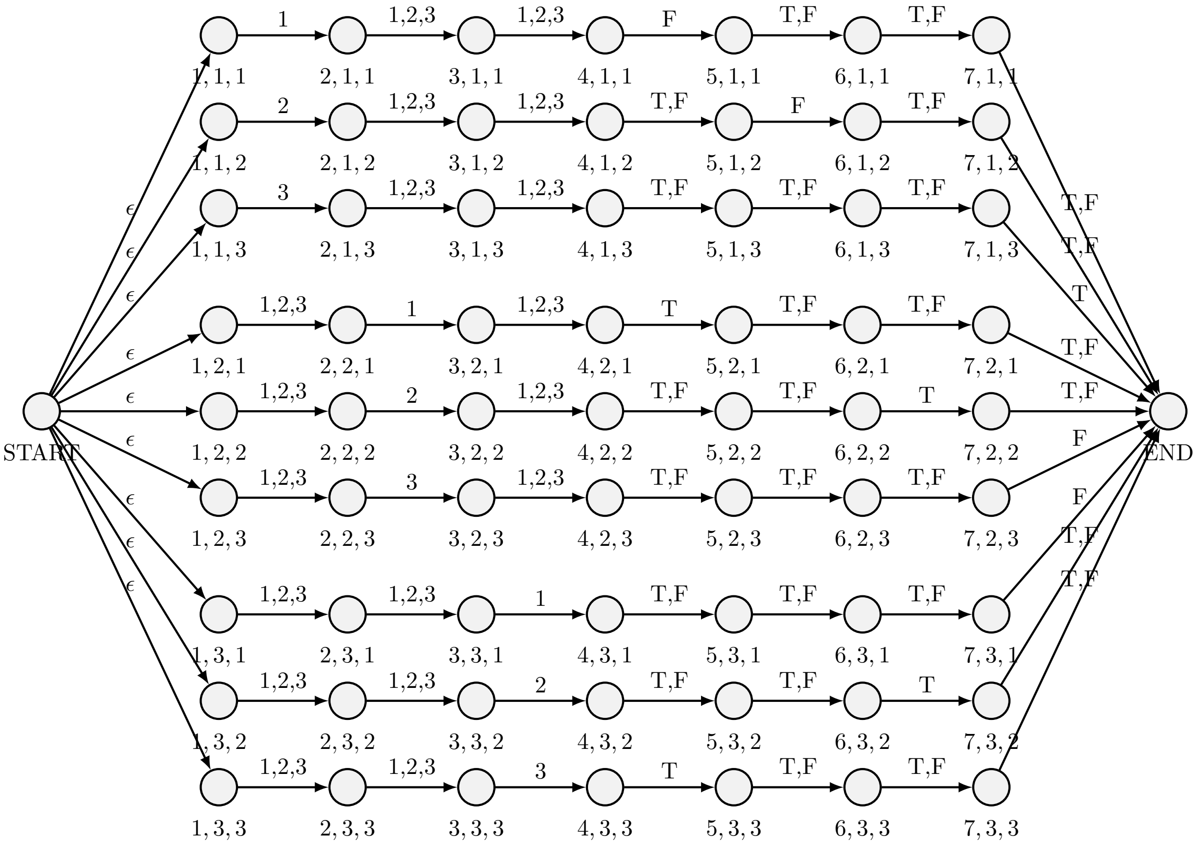


Figure 3.18: WFSA corresponding to the 3-Sat formula presented in Eq. (3.154).

Note A related problem, the Most Likely String in a WFSA Without Length, in which the length of the string is not explicitly defined, can be shown $\mathcal{NP}$-hard by a very similar proof. However, it cannot be shown $\mathcal{NP}$-complete by this proof, since the most likely string could be exponentially long, and thus the generate and test method cannot be used to show that the problem is in $\mathcal{NP}$.

3.8 Weighted Finite-State Transducers

Until now, we only looked at automata which *consume* input symbols and transition in the state space according to the transition function δ. We refered to those as finite state **acceptors**, alluding to the fact that they do not emit any symbols, but rather only follow instructions given by the input string. The terminology stems from the unweighted case where it is natural to say that a finite-state machine accepts or does not accept a string. We now introduce a another class of finite-state machines which, in addition to *consuming* a symbol, also *emit* one. Such automata are called finite-state **transducers** (FSTs). Transducer is the agentive of transduce, a high-brow English word for mapping one sequence to another. One could imagine a world where the inventors of FSTs chose a more intelligible—albeit perhaps more pedestrian—word for their new device, e.g., a finite-state mapper. But, alas, that is not our world, so we will stick with transducer in the class notes. Both finite-state acceptors and finite-state transducers are instances of finite-state automata, and, indeed, one can generalize the the two-tape transducers to n tapes if one so wishes. These are often called n-tape finite-state *machines* without any terminological flair. Intuitively, finite-state transducers are little more than finite-state automata in which each transition is augmented with an output symbol in addition to the familiar input symbol. As with the input symbols, the output symbols are concatenated along a path to form an output sequence. Weighted FSTs are finite-state transducers in which each transition carries a weight in addition to the input and output symbols, analogously to how WFSA augment FSA. We now define FSTs formally.

Definition 3.8.1 (Weighted Finite-state Transducer)**.** *A **weighted finite-state transducer** over a semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ is an* 8*-tuple* $(\Sigma, \Delta, Q, \delta, \lambda, \rho)$ *where*

- Σ *is a finite* input *alphabet;*
- Δ *is a finite* output *alphabet;*
- Q *is a finite set of states;*
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \times \mathbb{K} \times Q$ *a finite multi-set of transitions;*
- $\lambda : I \to \mathbb{K}$ *a weighting function over the set of initial states I;*
- $\rho : F \to \mathbb{K}$ *a weighting function over the set of final states F.*

We see the definition is very similar to the one of a WFSA. The only two differences are the new output alphabet Δ and the modification of the transition function δ to depend on *both input* and output *symbols.* Due to the similarities between acceptors and transducers, we will reuse a lot of the notation we developed on the former also on the latter. For example, the notion of a path remains very relevant here, with the only difference being that the transitions are now labeled with symbols *pairs.* The special symbols we saw in §3.1.1 are still relevant and have the same semantics.

We will only modify one definition and add one more. Recall that we use the notation $\Pi(\mathcal{Q}_1, \boldsymbol{y}, \mathcal{Q}_2)$ to denote a set of paths with symbols $\boldsymbol{y}$ in the case of WFSA. In the context of WFSTs, we now have 2 symbols for each path. As is turns out, in many applications of WFSTs, we are interested in possible *output strings* given an *input string.* Thus, it often makes sense to refer to the set of all paths with a certain input label. Therefore, we define the following additional overloadings of the path notation:

- $\Pi(\mathcal{Q}_1, \boldsymbol{y}, \mathcal{Q}_2)$ the set of paths in a WFST $\mathcal{T}$ with the *input label* $\boldsymbol{y}$;
- $\Pi(\mathcal{Q}_1, \boldsymbol{x}, \boldsymbol{y}, \mathcal{Q}_2)$ the set of paths with *input label* $\boldsymbol{x}$ and *output label* $\boldsymbol{y}$.

Now, we look at a few simple examples of WFSTs.

Example 3.8.1. *A simple example of a WFST $\mathcal{T}$ is shown in Fig. 3.19. Examples of pairs of strings accepted by $\mathcal{T}$ are $a : zz$, $abcda : xxwxwx$, and $abdcda : xxxwyyxwx$. Alternatively, we could for example say that $\mathcal{T}$ maps the string $abcda$ to the string $xxwxwx$.*

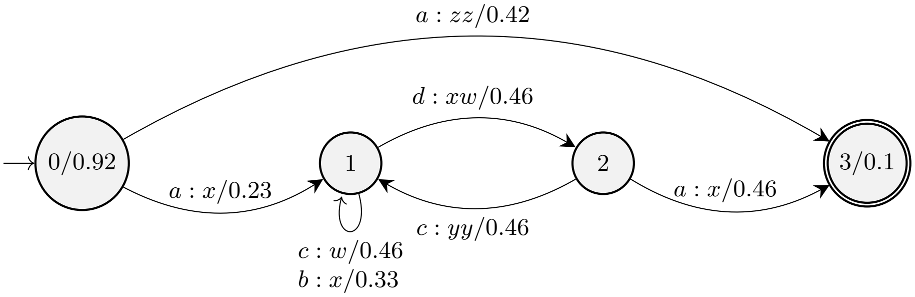


Figure 3.19: A simple WFST $\mathcal{T}$.

Example 3.8.2. *We can also encode WFSA as a WFST. An example of this is shown in Fig. 3.20. This adds no new information, but it is a convenient way of interpreting any acceptor as a transducer, and also allows us to use operations defined for transducers on acceptors as well. For example, composition, which is a generalization of intersection from §3.5.*

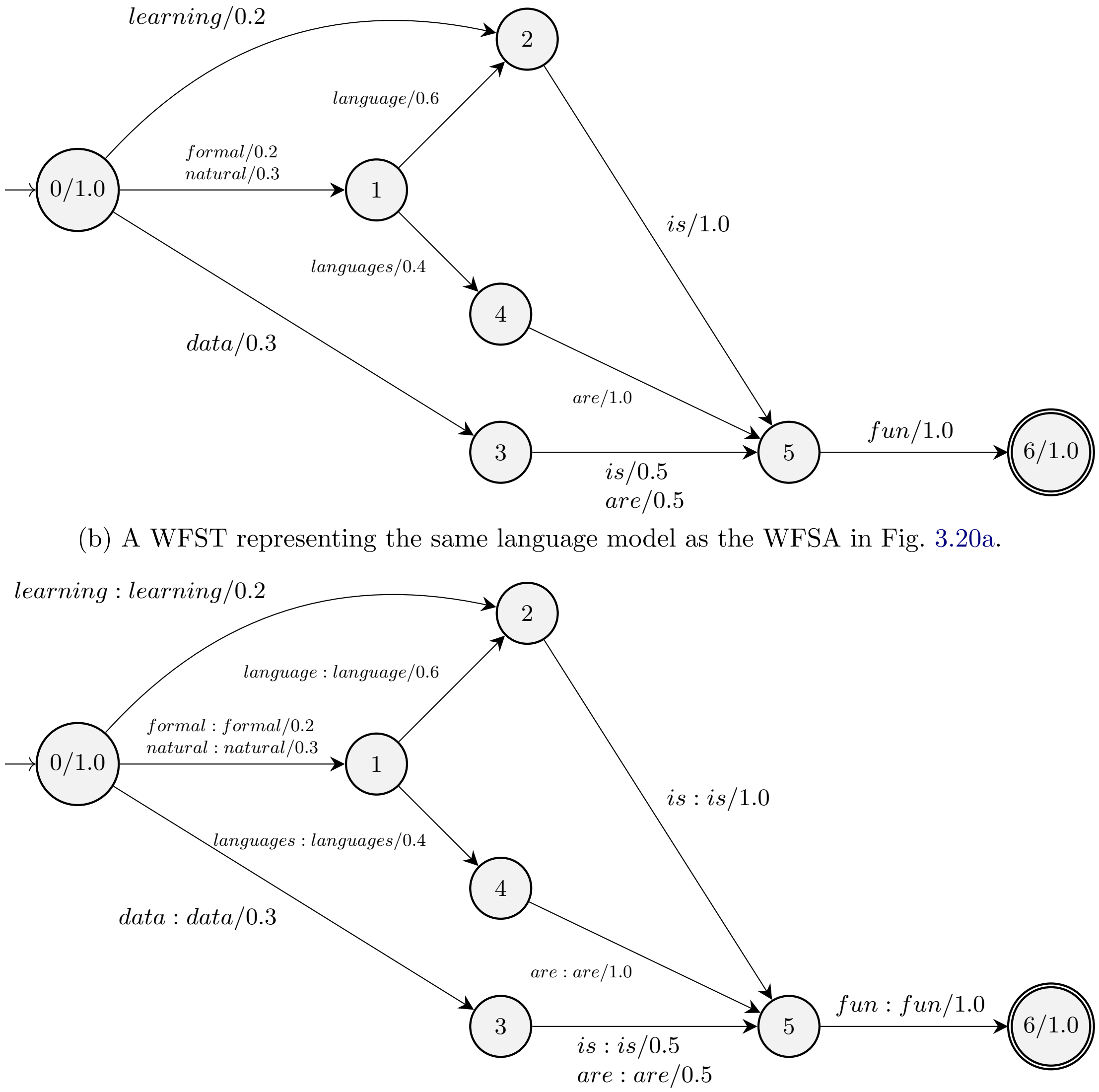
Example 3.8.3. *A common application of WFSTs is representation of* pronunciation lexicons*. There, they represent a mapping from phone sequences to words in the lexicon. The WFST in Fig. 3.21, taken from Mohri et al. (2008), shows a simple example for the words "data" and "dew", with probabilities representing the likelihoods of alternative pronunciations. Since a word pronunciation may be a sequence of several phones, the path corresponding to each pronunciation has ε-output labels on all but the word–initial transition. Since words are encoded by the output label, it is possible to combine the pronunciation transducers for more than one word without losing word identity.*

3.8.1 Encoding FSTs with the string semiring

We saw in Example 3.8.2 that we can trivially represent any WFSA as a WFST. The opposite is possible as well—we can encode an arbitrary WFST as a WFSA over a **product semiring** as we show shortly. But why would we want to do that? The ultimate motivation comes from the desire to determinize WFSTs, as we will see later in this lecture. The string semiring gives us a means to run the weighted determinize algorithm directly on a WFST. So, let's take a closer look at the **string semiring**, the first non-commutative semiring we will use in the course. It is formally defined as the tuple $\mathcal{S} = (\Sigma^* \cup \{\infty\}, \wedge, \circ, \infty, \varepsilon)$, where Σ is some alphabet and the two operations are defined as:

- $\wedge$: The longest common prefix (can be computed in linear time);
- $\circ$: Concatenation of two strings.

Exercise Exercise 3.10 asks you to show that the algebraic structure defined above is indeed a semiring. Importantly, it is a non-commutative semiring—the concatenation $\boldsymbol{x} \circ \boldsymbol{y}$ is obviously not the same as $\boldsymbol{y} \circ \boldsymbol{x}$. We make use of the element ∞ as the unit of the longest common prefix



(a) A WFSA $\mathcal{A}$ representing a (bi–gram) language model.

(b) A WFST representing the same language model as the WFSA in Fig. 3.20a.

Figure 3.20: An example of a WFSA encoded by a WFST.

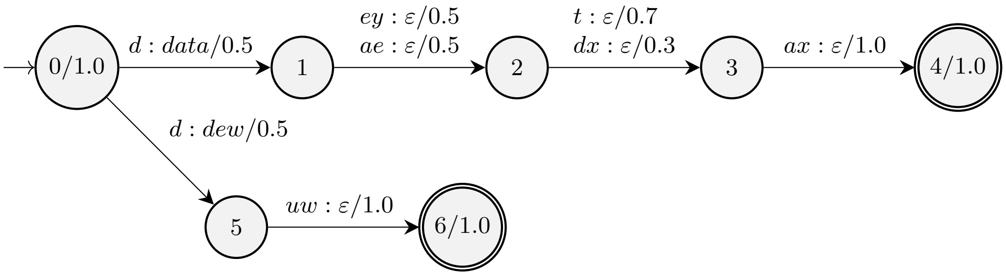


Figure 3.21: A WFST representing a simple pronunciation lexicon.

operation. ∞ is defined such that its longest common prefix with any string $\boldsymbol{y}$ is the string $\boldsymbol{y}$ itself and such that it satisfies the annihilation property, i.e, $\boldsymbol{y} \circ \infty = \infty \; \forall \boldsymbol{y} \in \Sigma^*$. We now also define the product semiring.

Definition 3.8.2 (Product Semiring)**.** *Let* $\mathcal{W}_1 = (\mathbb{K}_1, \oplus_1, \otimes_1, \boldsymbol{0}_1, \boldsymbol{1}_1)$ *and* $\mathcal{W}_2 = (\mathbb{K}_2, \oplus_2, \otimes_2, \boldsymbol{0}_2, \boldsymbol{1}_2)$ *be two semirings. The product semiring* $\mathcal{W} \overset{\text{def}}{=} \mathcal{W}_1 \times \mathcal{W}_2$ *is defined as* $(\mathbb{K}_1 \times \mathbb{K}_2, \oplus_\times, \otimes_\times, (\boldsymbol{0}_1, \boldsymbol{0}_2), (\boldsymbol{1}_1, \boldsymbol{1}_2))$*, where the* $\oplus_\times$ *and* $\otimes_\times$ *operations are the operations from the original semirings taken element-wise.*

Proposition 3.8.1. *The product semiring as defined in Def. 3.8.2 is a semiring.*

Proof. You are asked to show this in Exercise 3.13. ■

Let us now discuss how to use the product semiring to encode an arbitrary WFST as a WFSA. We will represent a WFST $\mathcal{T}$ over a semiring $\mathcal{W}$ as a WFSA $\mathcal{A}$ over the *product semiring* $\mathcal{S} \times \mathcal{W}$, where $\mathcal{S}$ is the string semiring. We can use the WFSA defined in such a way to translate a string over the input alphabet Σ into an output one from Δ—$\mathcal{A}$ takes an input string $\in \Sigma^*$ and traverses the state space while $\otimes$–multiplying the weights. As the $\otimes$ operation is taken in the product semiring, the weight tuple of any path accumulates the output string as the first element and the weight from the original transducer $\mathcal{T}$ as the second.

This is especially useful when one is explicitly interested in the weights of the input strings. In the WFSA representation of a WFST, we are not traversing the edges with pairs of symbols, but rather with only the input ones. This makes it possible to obtain the total weight of the input string using the general pathsum algorithms discussed in §3.6.1 as follows. Let $\mathcal{T} = (\Sigma, \Delta, Q, \delta, \lambda, \rho)$ be a WFST. Let $\mathcal{A}$ be its representation as an automaton over the product semiring. Then $\forall \boldsymbol{x} \in \Sigma^*$,

$$\mathcal{A}(\boldsymbol{x}) = (\boldsymbol{y}, w) \tag{3.155}$$

where

$$w = \bigoplus_{\substack{\boldsymbol{\pi} \in \Pi(\mathcal{T}), \\ \mathrm{s}\left(\boldsymbol{\pi}_{\text{input}}\right) = \boldsymbol{x}}} w(\boldsymbol{\pi}), \tag{3.156}$$

which is exactly the weight of the input string $\boldsymbol{x}$.

3.8.2 Composition of Weighted Finite-state Transducers

Composition is a general operation for combining two or more weighted transducers. It is very similar to intersection presented in §3.5 and is used to create a complex weighted transducer from simpler ones representing for example statistical or discriminative models. This section briefly introduces the operation; working out the details and implementing it is a Homework assignment. Formally, composition $\mathcal{T}_1 \circ \mathcal{T}_2$ of two WFSTs $\mathcal{T}_1 = (\Sigma, \Delta, Q_1, \delta_1, \lambda_1, \rho_1)$ and $\mathcal{T}_2 = (\Delta, \Xi, Q_2, \delta_2, \lambda_2, \rho_2)$ is the WFST $\mathcal{T} = (\Sigma, \Xi, Q, \delta, \lambda, \rho)$ such that

$$\mathcal{T}(\boldsymbol{x}, \boldsymbol{y}) = \bigoplus_{\boldsymbol{z} \in \Delta^*} \mathcal{T}_1(\boldsymbol{x}, \boldsymbol{z}) \otimes \mathcal{T}_2(\boldsymbol{z}, \boldsymbol{y}). \tag{3.157}$$

Notice that the *output* alphabet of the first automaton matches the *input* alphabet of the second one. The input alphabet of the resulting transducer $\mathcal{T}$ is the input alphabet Σ of $\mathcal{T}_1$ while $\mathcal{T}$'s output alphabet is the output alphabet Ξ of $\mathcal{T}_2$. Here, we assume that the infinite sum in Eq. (3.157) is well defined.

Let us discuss the intuition one should have about the weight associated with a string pair $\boldsymbol{x}, \boldsymbol{y}$ in the output transducer. As Eq. (3.157) expresses, it is a sum over all $\boldsymbol{z} \in \Delta$ where each term is the $\otimes$-multiplication of the weights given to the pairs $\boldsymbol{x}, \boldsymbol{z}$ by $\mathcal{T}_1$ and $\boldsymbol{z}, \boldsymbol{y}$ by $\mathcal{T}_2$. You can therefore think of it as the weight of all possible "intermediary" strings $\boldsymbol{z}$ such that $\mathcal{T}_1$ produces $\boldsymbol{z}$ from $\boldsymbol{x}$ and $\mathcal{T}_2$ transforms $\boldsymbol{z}$ into $\boldsymbol{y}$. In other terms, it is intuitively the $\oplus$-sum of the weights all the possible transformations $\boldsymbol{x} \overset{\mathcal{T}_1}{\to} \boldsymbol{z} \overset{\mathcal{T}_2}{\to} \boldsymbol{y}$, where the weight of a complete 2-stage transformation is obtained by $\otimes$-multiplying the weights of the individual transformations.

Intersection is actually a *special case* of composition, in the sense that intersection of two acceptors is composition performed on transducers constructed from them. The algorithm for composition therefore has a very similar form to the one for intersection, with the main difference being the fact we are not matching in the *input* symbols of two acceptors, but rather on the *ouput symbol* of $\mathcal{T}_1$ and the *input symbol* of $\mathcal{T}_2$.

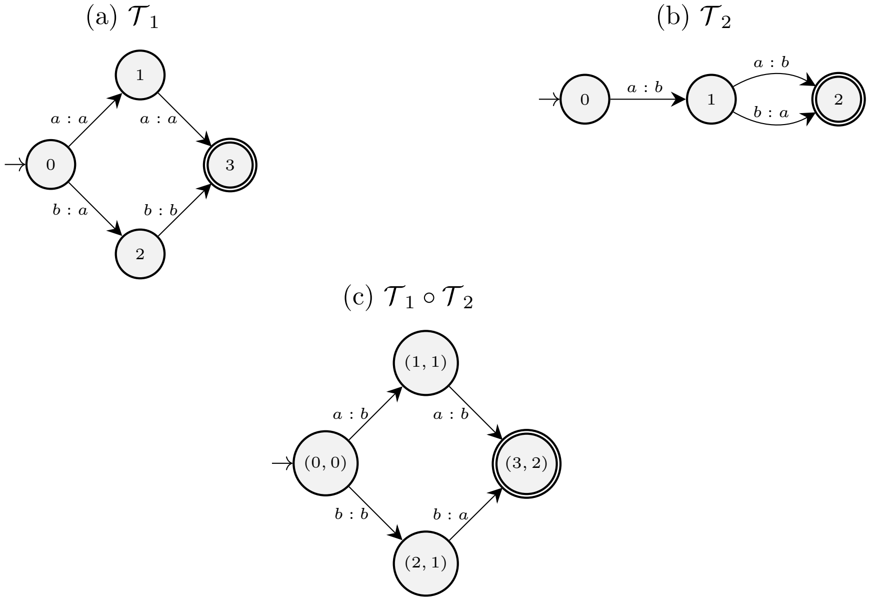
Since intersection is just a special case of composition, the latter suffers from the same problem of handling ε-transitions. In the case of transducers, the problem arises whenever $\mathcal{A}_1$ *emits* and $\mathcal{A}_2$ *consumes* ε's. The issue is handled with the same ε-filter as was introduced in §3.5—the composition we use in the augmented intersection now is just the composition of the augmented input automata and the filter!

Algorithm 15 The naïve version of the algorithm for computing the composition of two WFSTs.

```
def NaïveComposition($\mathcal{T}_1, \mathcal{T}_2$):
    $\mathcal{T} \leftarrow (\Sigma, \Delta, Q, \delta, \lambda, \rho)$        ▷Create a new WFST
    for $q_1, q_2 \in Q_1 \times Q_2$ :
        for $q_1 \xrightarrow{a:b/w_1} q_1', q_2 \xrightarrow{c:d/w_2} q_2' \in \mathcal{E}_{\mathcal{A}_1}(q_1) \times \mathcal{E}_{\mathcal{A}_2}(q_2)$ :
            if $b = c$ :
                add states $(q_1, q_2)$ and $(q_1', q_2')$ to $\mathcal{A}$ if they have not been added yet
                add arc $(q_1, q_2) \xrightarrow{a:d/w_1 \otimes w_2} (q_1', q_2')$ to $\mathcal{A}$
    for $(q_1, q_2) \in Q$ :
        $\lambda_{\mathcal{A}} \leftarrow \lambda_1(q_1) \otimes \lambda_2(q_2)$
        $\rho_{\mathcal{A}} \leftarrow \rho_1(q_1) \otimes \rho_2(q_2)$
    return $\mathcal{A}$
```

Algorithm 16 On-the-fly (accessible) WFST composition.

```
def Intersect($\mathcal{T}_1, \mathcal{T}_2$):
    stack $\leftarrow [(q_1, q_2) \mid q_1 \in I_1, q_2 \in I_2]$
    visited $\leftarrow \{(q_1, q_2) \mid q_1 \in I_1, q_2 \in I_2\}$
    $\mathcal{T} \leftarrow (\Sigma, \Delta, Q, \delta, \lambda, \rho)$                ▷Create a new WFST
    while |stack| $> 0$ :
        $q_1, q_2 \leftarrow$ stack.pop()
        for $q_1 \xrightarrow{a:b/w_1} q_1', q_2 \xrightarrow{c:d/w_2} q_2' \in \mathcal{E}_{\mathcal{A}_1}(q_1) \times \mathcal{E}_{\mathcal{A}_2}(q_2)$ :
            if $b = c$ :
                add states $(q_1, q_2)$ and $(q_1', q_2')$ to $\mathcal{T}$ if they have not been added yet
                add arc $(q_1, q_2) \xrightarrow{a:d/w_1 \otimes w_2} (q_1', q_2')$ to $\mathcal{A}$
                if $(q_1', q_2')$ not in visited :
                    push $(q_1', q_2')$ to stack and visited
    for $(q_1, q_2) \in Q$ :
        $\lambda_{\mathcal{T}} \leftarrow \lambda_1(q_1) \otimes \lambda_2(q_2)$
        $\rho_{\mathcal{T}} \leftarrow \rho_1(q_1) \otimes \rho_2(q_2)$
    return $\mathcal{T}$
```



(a) $\mathcal{T}_1$

(b) $\mathcal{T}_2$

(c) $\mathcal{T}_1 \circ \mathcal{T}_2$

Figure 3.22: An example of on-the-fly composition of two WFSTs.

3.9 Weighted ε-Removal

We now start our deep dive into algorithms that transform WFSAs, i.e., algorithms which take as input an automaton $\mathcal{A}$ and produce from it an automaton $\mathcal{A}'$ that is equivalent in a certain sense. We are generally interested in preserving the string sums of the automaton $\mathcal{A}$, but adding additional desirable properties. This chapter and the following introduce weighted ε-removal and weighed determinization, together with the assumptions we must make on the semiring to define the transformations. We start by defining the identity of finite-state automata, both in the unweighted and weighted case.

Algorithm 17 The on-the-fly version of the algorithm for computing the composition of two WFSTs with epsilon filter. The ε-FILTER function is the interface to the filter transducer $\mathcal{T}_{\mathcal{F}}$. ε-FILTER (a_1, a_2, q_f) returns the target state of the transition from q_f with the label $a_1 : a_2$ in $\mathcal{T}_{\mathcal{F}}$.

```
def $\epsilon$-filterCompose$(\mathcal{T}_1, \mathcal{T}_2)$:
    $\mathcal{T} \leftarrow (\Sigma, \Delta, Q, \delta, \lambda, \rho)$        ▷Initialize a new WFST over the same semiring as $\mathcal{T}_1$ and $\mathcal{T}_2$.
    stack $\leftarrow \{(q_1, q_2, 0) \mid q_1 \in I_1, q_2 \in I_2\}$
    visited $\leftarrow \{(q_1, q_2, 0) \mid q_1 \in I_1, q_2 \in I_2\}$
    ▷ Rename $\varepsilon$-transitions in $\mathcal{A}_1$ and $\mathcal{A}_2$
    rename edges $q \xrightarrow{a:\varepsilon/w} q'$ to $q \xrightarrow{a:\varepsilon_2/w} q'$ in $\mathcal{T}_1$
    rename edges $q \xrightarrow{\varepsilon:b/w} q'$ to $q \xrightarrow{\varepsilon_1:b/w} q'$ in $\mathcal{T}_2$
    ▷ Add self loops
    for $q_1 \in Q_1$ :
        add edge $q_1 \xrightarrow{\varepsilon:\varepsilon_1/\mathbf{1}} q_1$ to $\delta_1$
    for $q_2 \in Q_2$ :
        add edge $q_2 \xrightarrow{\varepsilon_2:\varepsilon/\mathbf{1}} q_2$ to $\delta_2$
    ▷ We refer to the augmented input transducers with (and their components) with $\widetilde{\mathcal{T}_1}$ and $\widetilde{\mathcal{T}_2}$
    ▷ Body of the algorithm
    while |stack| $> 0$ :
        $q_1, q_2, q_f \leftarrow$ stack.pop()
        for $q_1 \xrightarrow{a:b/w_1} q_1', q_2 \xrightarrow{c:d/w_2} q_2' \in \widetilde{\delta}_1 \times \widetilde{\delta}_2$ :
            if $\varepsilon$-FILTER $(b, c, q_f) \neq \perp$ :
                $q_f' \leftarrow \varepsilon$-FILTER $(q_f, a_1, a_2)$
                add arc $(q_1, q_2, q_f) \xrightarrow{a_1/w_1 \otimes w_2} (q_1', q_2', q_f')$ to $\mathcal{T}$
                if $\left(q_1', q_2', q_f'\right) \notin$ visited :
                    push $\left(q_1', q_2', q_f'\right)$ to stack
                    push $\left(q_1', q_2', q_f'\right)$ to visited
    ▷ Adding initial and final weights
    for $(q_1, q_2, q_f) \in Q$ :
        $\lambda(q_1, q_2, q_f) \leftarrow \widetilde{\lambda}_1(q_1) \otimes \widetilde{\lambda}_2(q_2)$
        $\rho(q_1, q_2, q_f) \leftarrow \widetilde{\rho}_1(q_1) \otimes \widetilde{\rho}_2(q_2)$
    return $\mathcal{A}$
```

Definition 3.9.1 (Language and State Identity)**.** *Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be a FSA. Two states $q, q' \in Q$ are **identical** if and only if their respective **right languages** are the same. The right language of a state q is the set of words that are accepted if we treat q as if it were the only initial state in $\mathcal{A}$.*

In the unweighted case, the identity considered the accepted languages. However, in the case of weighted FSA, we must also take into account the weights.

Definition 3.9.2 (Weighted Language Identity)**.** *Two weighted languages are **identical** if the formal power series representing them are the same, meaning that they assign the same weights to all $\boldsymbol{y} \in \Sigma^*$.*

Definition 3.9.3 (Weighted Identity of States)**.** *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA. Two states $q, q' \in Q$ **identical** if their right languages are the same. The right language of a state q in a weighted automaton is the* weighted *language of the WFSA in which q is the sole initial state with the initial weight $\lambda(q)$ if $q \in I$ or $\boldsymbol{1}$ otherwise. We will denote the right language of a state q as L_q. See Quint (2004) for further discussion for identity in the* weighted *case.*

Definition 3.9.4 (Automata Identity)**.** *Two (weighted) automata are **identical** if and only if all their initial states are identical.*

We will also use the term equivalent to describe two identical automata, but the distinction between the two terms will be very important when talking about states, as we will see in the next chapter.

Informally, you can think of the right languages of a state q as the (weighted) language of strings starting in q and ending in some final state. Right languages will be crucial for our discussion of weighted minimization in §3.11.6.

Identity of two WFSA does not mean that the weight distribution of the two automata is the same—indeed, all the weights can be different, yet the automata are still equivalent. As we will see, weight pushing, which we introduce in the next chapter, deals with reweighting the edges of an automaton such that the weights are assigned in a canonical fashion.

We now have the tools to talk about ε-removal. There are many different places in finite-state automata theory were we intentionally insert ε-transitions. For instance, the most natural construction for computing the Kleene closure of an FSA involves ε-transitions and the most natural construction for computing the union of two FSAs involves ε-transitions. The reason for this is that it's often easier to specify certain constructions using ε-transitions. However, how would we go about removing ε from these constructions? Moreover, as we will see, the determinization algorithms we introduce later in this chapter assume the input automata are ε-free.

At first blush, one might think that we could simply delete the ε-transitions from the WFSA. Further thought, however, reveals that such a strategy obviously not work out—the resulting automaton would only accept a subset of the strings accepted by the original one. Instead, the idea is to *directly connect* a given state q to the states q'' which are reachable from q by taking *exactly one non-ε-transition* to some state q' followed by any number of ε-transitions. These new transitions from q to q'' have to be weighted with the weight of all the ε-yielding paths from q' to q'' to retain the same weighting of strings—they are therefore weighted with the $\oplus$-sum of the weights of the ε-only paths $\otimes$-multiplied with the weight of the final non-ε-transition to the state. You can think of this reweighting as a way to explicitly account for all possible ε jumps after consuming exactly one non-ε-transition from q. This is illustrated in Fig. 3.23.

The crux behind the ε-removal algorithms is the computation of the **ε-closure**. In the unweighted case, the ε-closure of a state $q \in Q$ is set of the all states one can reach from q while *only* taking

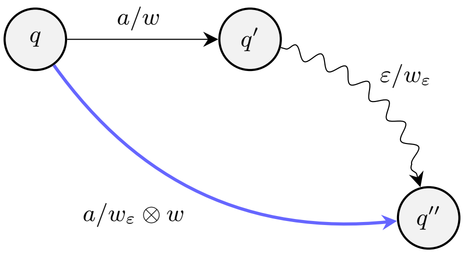


Figure 3.23: A schema of how ε-removal must work. The curly edge marks an ε-labeled path of arbitrary length. We assume the blue edge between q and q'' is not present in the original WFSA and is *created* during ε-removal. If there is an edge between q and q'' beforehand, a new one is created, or, alternatively, the weight $w_\varepsilon \otimes w$ is $\oplus$-added to the original weight.

ε-transitions. In the weighted case, the set, i.e, a binary inclusion–exclusion decision, is relaxed to a weight in the semiring. The following definition formalizes this idea.

Definition 3.9.5. *The weighted* ε***-closure*** *of a WFSA* $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ *over semiring* $\mathcal{W}$ *is*

$$\kappa(q, q') = \bigoplus_{\boldsymbol{\pi} \in \Pi(q,\varepsilon,q')} \bigotimes_{p \xrightarrow{\varepsilon/w} p' \in \boldsymbol{\pi}} w \tag{3.158}$$

If $\mathcal{A}$ *has an* ε*-cycle, we require* $\mathcal{W}$ *to be closed and have a well-defined pathsum. The intuitive interpretation of Eq.* (3.158) *is sum over all paths from q to* q' *with yield exactly* ε*.*

Computing the ε-closure Computation of ε-closure of the automaton $\mathcal{A}$ is done in two steps.

(i) We construct the automaton $\mathcal{A}_\varepsilon$, which is a copy of $\mathcal{A}$ keeping *only* the ε-transitions in $\mathcal{A}$;

(ii) We compute the Kleene closure of $\mathcal{A}_\varepsilon$'s transition matrix using Lehmann's algorithm.

It is easy to see that the second step will produce exactly the pathsums of paths with the yield ε between all states of the original $\mathcal{A}$, which is exactly the ε-closure. The full algorithm for ε-removal is outlined in Alg. 18. The implementation is very simple, since most of the work in the algorithm is done by Lehmann's during the computation of the weighted ε-closure of $\mathcal{A}$. Line 6 captures the intuition given in Fig. 3.23. The weight of the edge $q \xrightarrow{a/\bullet} q''$ gets increased by the value of the epsilon closure of the states q' (which is reachable from q in one hop with a a-transition) and q'' (which has a particular value of the ε-closure with q') $\otimes$-multiplied with the weight of the a-transition between q and q'. Said differently, the weight of the edge $q \xrightarrow{a/\bullet} q''$ is increased by the weight of the ε-paths between q' and q'', since you can, after taking the a-transition, always hop from q' to q'' using ε-transtions.

We will now prove the correctness of the ϵ-removal algorithm. We will make use of the following lemma for that.

Lemma 3.9.1. *Let* $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ *be a WFSA over semiring* $\mathcal{W}$*. If* $\mathcal{A}$ *has an* ε *cycle, then we require* $\mathcal{W}$ *to be closed and have a well-defined pathsum. The output* $\mathcal{A}_{\neg\varepsilon} =$ ϵ-removal$(\mathcal{A})$ *of* ϵ-removal *preserves the forward weights, i.e.,* $\boldsymbol{\alpha}_{\mathcal{A}}(\boldsymbol{y}, q) = \boldsymbol{\alpha}_{\mathcal{A}_{\neg\varepsilon}}(\boldsymbol{y}, q)$ *for all* $\boldsymbol{y} \in \Sigma^*$*.*

Proof. $\mathcal{A}_{\neg\varepsilon}$ is ε-free by construction. We prove equivalence by induction on the length of $\boldsymbol{y}$.

Algorithm 18 The ϵ-removal algorithm.

```
1.  def $\epsilon$-removal($\mathcal{A}$):
2.      compute the $\varepsilon$-closure $\kappa(\cdot,\cdot)$    ▷Requires Lehmann's algorithm; runs in $\mathcal{O}\left(|Q|^3\right)$
3.      initialize $\mathcal{A}_{\neg\varepsilon}$ as a copy of $\mathcal{A}$ without $\varepsilon$-transitions    ▷Final weights copied, initial weights not
4.      for $q \in Q$ :
5.          for $q \xrightarrow{a/w} q' \in \mathcal{E}(q)$ :
6.              for $q'' \in Q$ :
7.                  add $w \otimes \kappa(q', q'')$ to edge $q \xrightarrow{a} q''$'s current weight in $\mathcal{A}_{\neg\varepsilon}$
8.      for $q' \in Q$ :
9.          $\lambda'(q') \leftarrow \bigoplus_{q \in Q} \lambda(q) \otimes \kappa(q, q')$
10.     return $\mathcal{A}_{\neg\varepsilon}$
```

Base case: $\boldsymbol{y} = \varepsilon$ ($|\boldsymbol{y}| = 0$). Since $\mathcal{A}_{\neg\varepsilon}$ is ε-free, the forward weight of the string ε is:

$$\begin{aligned}
\boldsymbol{\alpha}_{\mathcal{A}_{\neg\varepsilon}}(\boldsymbol{y}, q') &= \boldsymbol{\alpha}_{\mathcal{A}_{\neg\varepsilon}}(\varepsilon, q') && \text{(definition of } \boldsymbol{y}\text{)} && (3.159)\\
&= \lambda'\left(q'\right) && \text{(definition of } \boldsymbol{\alpha}_{\mathcal{A}_{\neg\varepsilon}}(\varepsilon)\text{)} && (3.160)\\
&= \bigoplus_{q \in Q} \lambda(q) \otimes \kappa(q, q') && \text{(Alg. 18 Line 9)} && (3.161)\\
&= \bigoplus_{q \in Q} \lambda(q) \otimes \bigoplus_{\boldsymbol{\pi} \in \Pi(q, \varepsilon, q')} w(\boldsymbol{\pi}) && \text{(definition of } \kappa(\cdot,\cdot)\text{)} && (3.162)\\
&= \boldsymbol{\alpha}_{\mathcal{A}}\left(\varepsilon, q'\right) && \text{(definition of } \boldsymbol{\alpha}_{\mathcal{A}}\text{)} && (3.163)
\end{aligned}$$

Inductive Step. We now assume $\boldsymbol{\alpha}_{\mathcal{A}}(\boldsymbol{y}, q) = \boldsymbol{\alpha}_{\mathcal{A}_{\neg\varepsilon}}(\boldsymbol{y}, q)$ holds for any $\boldsymbol{y}$ with $|\boldsymbol{y}| < n$. Let $\boldsymbol{y} \in \Sigma^*$ such that $|\boldsymbol{y}| = n - 1$ and $b \in \Sigma$. Here, we use $w_{\mathcal{A}}(\boldsymbol{\pi})$ to denote the weight assigned to path $\boldsymbol{\pi}$ by the automaton $\mathcal{A}$ and $w_{\mathcal{A}_{\neg\varepsilon}}(\boldsymbol{\pi})$ analogously. The result follows by manipulation as follows:

$$\begin{aligned}
\boldsymbol{\alpha}_{\mathcal{A}}(\boldsymbol{y} \circ b, q') &= \bigoplus_{q \in Q} \boldsymbol{\alpha}_{\mathcal{A}}(\boldsymbol{y}, q) \otimes \bigoplus_{q \xrightarrow{b/w} q'' \in Q} w \otimes \kappa(q'', q') && \text{(definition)} && (3.164)\\
&= \bigoplus_{q \in Q} \boldsymbol{\alpha}_{\mathcal{A}_{\neg\varepsilon}}(\boldsymbol{y}, q) \otimes \bigoplus_{q \xrightarrow{b/w} q'' \in Q} w \otimes \kappa(q'', q') && \text{(inductive hypothesis)} && (3.165)\\
&= \bigoplus_{q \in Q} \boldsymbol{\alpha}_{\mathcal{A}_{\neg\varepsilon}}(\boldsymbol{y}, q) \otimes w_{\mathcal{A}_{\neg\varepsilon}}(q \xrightarrow{b} q') && \text{(Alg. 18 Line 7)} && (3.166)\\
&= \boldsymbol{\alpha}_{\mathcal{A}_{\neg\varepsilon}}(\boldsymbol{y} \circ b, q') && \text{(definition)} && (3.167)
\end{aligned}$$

The first line of the proof follows from the definition of the forward weights as the sum of all the path weighs with yield $\boldsymbol{y} \circ b$. We can decompose that sum into the sum of all the forward weights to some $q \in Q$ multiplied by the sum of the paths from q to q' labeled with ε including the possible ε-transitions. ■

Theorem 3.9.1. *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA over semiring $\mathcal{W}$. If $\mathcal{A}$ has an ε cycle, then we require $\mathcal{W}$ to be closed and have a well-defined pathsum. Then, $\mathcal{A}_{\neg\varepsilon}$ is equivalent to $\mathcal{A}$.*

Proof. For an arbitrary $\boldsymbol{y} \in \Sigma^*$, the following manipulation proves the result:

$$\mathcal{A}_{\neg\varepsilon}(\boldsymbol{y}) = \bigoplus_{q \in Q} \boldsymbol{\alpha}_{\mathcal{A}_{\neg\varepsilon}}(q) \otimes \rho_{\mathcal{A}_{\neg\varepsilon}}(q) \qquad \text{(definition of string weight)} \tag{3.168}$$

$$= \bigoplus_{q \in Q} \boldsymbol{\alpha}_{\mathcal{A}}(q) \otimes \rho_{\mathcal{A}_{\neg\varepsilon}}(q) \qquad \text{(Lemma 3.9.1)} \tag{3.169}$$

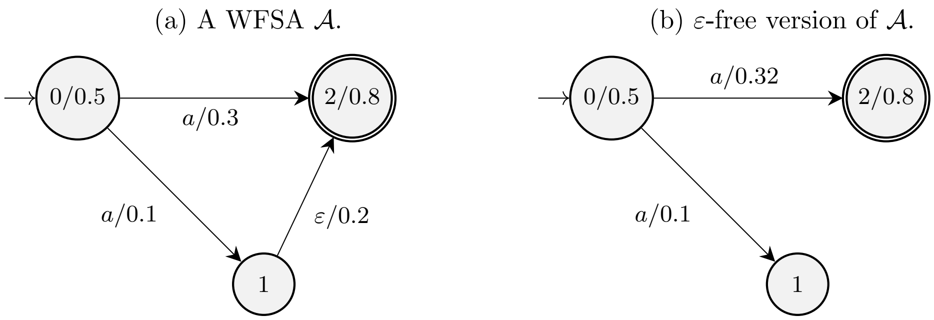
$$= \bigoplus_{q \in Q} \boldsymbol{\alpha}_{\mathcal{A}}(q) \otimes \rho_{\mathcal{A}}(q) \qquad \text{(Alg. 18 Line 3)} \tag{3.170}$$

$$= \mathcal{A}(\boldsymbol{y}) \qquad \text{(definition of string weight)} \tag{3.171}$$

The third line follows from the fact that the final weights were directly copied into $\mathcal{A}_{\neg\varepsilon}$ from $\mathcal{A}$ without modification. ■

The algorithm ϵ-removal does not change the state space of the original automaton. However, some states may become *useless* after any application of ϵ-removal: Specifically, any state with only incoming ε-transitions after adding the new edges (and before removing ε-transitions) becomes non-accessible, while any state with only ε outgoing transitions becomes non-coacessible. The non-accessible and non-co-accessible states can be removed through trimming, and it is generally recommended that a call to Alg. 18 be followed by a call to **??**. It can also be the case that ϵ-removal creates multiple transitions between the same state pair with the same labels—in this case, to avoid needless replication of edges, the weights of these transitions are $\oplus$-summed and turned into a single transitions, which can be done in $\mathcal{O}(|E|)$ time.

Example 3.9.1 (ε-removal)**.** *A simple example of ε-removal is presented in Fig. 3.24. Notice that the algorithm would originally construct additional a-transition between state* 0 *and* 2 *with the weight* $0.2 \cdot 0.1 = 0.02$. *However, we can simply add the weight to the existing a-transition to obtain a single transition with the weight* 0.32. *Moreover, notice that the state* 1 *becomes* non-co-accessible *in the ε-free automaton, since its only path to a final state in the original automaton yielded ε and was thus removed. Its weight was directly incorporated into the transition from* 0 *to* 2.



(a) A WFSA $\mathcal{A}$.

(b) ε-free version of $\mathcal{A}$.

Figure 3.24: An example of ε-removal on a simple automaton $\mathcal{A}$ over the real semiring.

3.10 Determinization

In this section we are going to discuss determiniztion, a procedure that removes non-determinism from a WFSA as the name of the algorithm suggests. Determinization has the effect of allowing one to obtain the weight of a string in linear time *without* the need to sum over paths. In the unweighted case, determinization is always possible; however, as we will see, determinization is *not* always possible in the weighted case. Fortunately, under in certain semirings, determinizability of an automaton can be tested with a conceptually simple algorithm.

3.10.1 Unweighted Determinization

Def. 3.9.1 gives us the vocabulary we need to discuss determinization. We will start our exposition of it at an intuitive level. Fundamentally, non-determinism is the fact that a WFSA can be in more than one state at once; non-determinism arises due to the fact that multiple transitions can be taken at once in some cases. Intuitively, to determinize a WFSA, then, is to make it such that the machine is in at most one state at a time. The crucial insight behind all determinization algorithms we will present is this: Since we have a *finite* set of state machine, there are a *finite* number of subsets of states we can be in at any time. Indeed, this insight leads suggests a simple brute-force algorithm known as the powerset construction: We enumerate all possible subset of states we could be in and connect them to each other. More formally, given an ε-free FSA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, our goal is to construct a new machine $\mathcal{A}_{\text{DET}}$ that operates on the powerset 2^Q. We refer to elements of 2^Q as **power state** $\mathcal{Q} \subseteq Q$ and denote them using a distinguished calligraphic font. However, this notational switch is just for expositional purposes to make it easier to understand that a state in a determinized machine corresponds to a *set* of states in the machine that is being determinized. To finish the intuition behind the power set construction, we add edges to the machine in the following fashion: For every pair of power states $\mathcal{Q} = \{q_1, \ldots\}$ and $\mathcal{Q}' = \{q'_1, \ldots\}$, there exists an edge $\mathcal{Q} \xrightarrow{a} \mathcal{Q}'$ in $\mathcal{A}_{\text{DET}}$ if and only if there exists a $q_n \in \mathcal{Q} = \{q_1, \ldots\}$ and a $q'_m \in \mathcal{Q}' = \{q'_1, \ldots\}$ with edge $q_n \xrightarrow{a} q'_m$.

However, the powerset construction is woefully inefficient. While it is true that the asymptotic worse-case of any determinization must be exponential in the size of the input machine, the brute-force algorithm suggested above *always* runs in exponential time. This is undesirable. Thus, we will develop an on-the-fly determinization method that considers only those elements of the powerset of states that are strictly necessary. At a high level, we proceed by combining an accessibility check with the powerset construction. This is similar to the on-the-fly version of the intersection algorithm from the last chapter in that we ensure that we only visit accessible states by keeping a stack of states that could be reached in the resulting automaton if we start in its starting state. These states are visited (expanded) sequentially, and new power states are added to the stack and the resulting automaton by constructing the set of states which can be reached from the current power state by following the same symbol. The whole algorithm is outlined in Alg. 19.

Hopefully it is clear that Alg. 19 is a direct formalization of the textual description written above. Let us try to understand what the algorithm is doing at a deeper level. Since $\mathcal{A}$ can be in any of the initial states at the beginning, the initial power state of $\mathcal{A}_{\text{DET}}$ is $\{I\}$, i.e., the power state of all initial states in $\mathcal{A}$. The determinized machine's power state is put on the queue `stack` of unexplored states which are expanded sequentially. The queue keeps unexpanded accessible power states. Expansion of a power state $\mathcal{Q}$ refers to the processing of all outgoing transitions of $\mathcal{Q}$ where we create new accessible power states reachable over the processed transitions or we connect $\mathcal{Q}$ to existing power states reachable over them. Thus, once a power state has been expanded, all its outgoing arcs are included in the output automaton. More precisely, any time a power state $\mathcal{Q} \in Q_{\text{DET}}$ is popped from the stack on Line 5, all the states $q \in Q$ which can be reached from any

Algorithm 19 The Determinize algorithm. The symbols function returns the symbols of the transitions in the input set of transitions.

```
1.  def Determinize($\mathcal{A}$):
2.    $\mathcal{A}_{\text{DET}} \leftarrow (\Sigma, Q, I, F, \delta)_{\text{DET}}$                ▷Initialize a new FSA
3.    stack $\leftarrow [\mathcal{Q}_I]$        ▷Initialize the stack to contain the initial states of the input automaton
4.    $Q_{\text{DET}} \leftarrow \{\mathcal{Q}_I\}$                ▷Add the initial state to $Q_{\text{DET}}$
5.    while |stack| $> 0$ :
6.      pop power state $\mathcal{Q}$ from stack
7.      for $a \in$ symbols $(\mathcal{E}_{\mathcal{A}}(\mathcal{Q}))$ :        ▷Consider all outgoing transition symbols.
8.        $\mathcal{Q}' \leftarrow \left\{q : \bullet \xrightarrow{a} q \in \mathcal{E}_{\mathcal{A}}(\mathcal{Q})\right\}$        ▷Construct a new power state from all the states reachable with a.
9.        $\delta_{\text{DET}} \leftarrow \delta_{\text{DET}} \cup \left\{\mathcal{Q} \xrightarrow{a} \mathcal{Q}'\right\}$
10.       if $\mathcal{Q}' \notin Q_{\text{DET}}$ :
11.         $Q_{\text{DET}} \leftarrow Q_{\text{DET}} \cup \{\mathcal{Q}'\}$                ▷Add new states to $Q_{\text{DET}}$
12.         if $\mathcal{Q}' \cap F \neq \emptyset$ :
13.           $F_{\text{DET}} \leftarrow F_{\text{DET}} \cup \{\mathcal{Q}'\}$
14.         push $\mathcal{Q}'$ to stack                ▷Push the new power state to the stack.
15.   return $\mathcal{A}_{\text{DET}}$
```

of the states $q \in \mathcal{Q}$ with the *same symbol* are combined into a new power state $\mathcal{Q}'$ (Line 8). This new power state is then push to the stack of unexpanded states. Intuitively, we can see that only power states reachable from I are processed—this means that $\mathcal{A}_{\text{DET}}$ only has accessible states. And, importantly, we can see that $\mathcal{A}_{\text{DET}}$ is deterministic by construction: There are no ε-transitions and we have only one outgoing edge for symbol for every power state $\mathcal{Q}$ in $\mathcal{A}_{\text{DET}}$. Since the original automaton $\mathcal{A}$ is in a final state any time it can reach one of the final states with a certain string (again, recall that the automaton implicitly takes all possible transitions), all the power states which contain a final state are added to the final set of states of $\mathcal{A}_{\text{DET}}$. In the unweighted case, it is clear that the algorithm terminates. Every state $\mathcal{Q}$ in $\mathcal{A}$ is identified with an element of 2^Q. Thus, only a finite number of states $\mathcal{Q}$ are pushed onto the stack and loop at Line 4 iterates only a finite number of times.

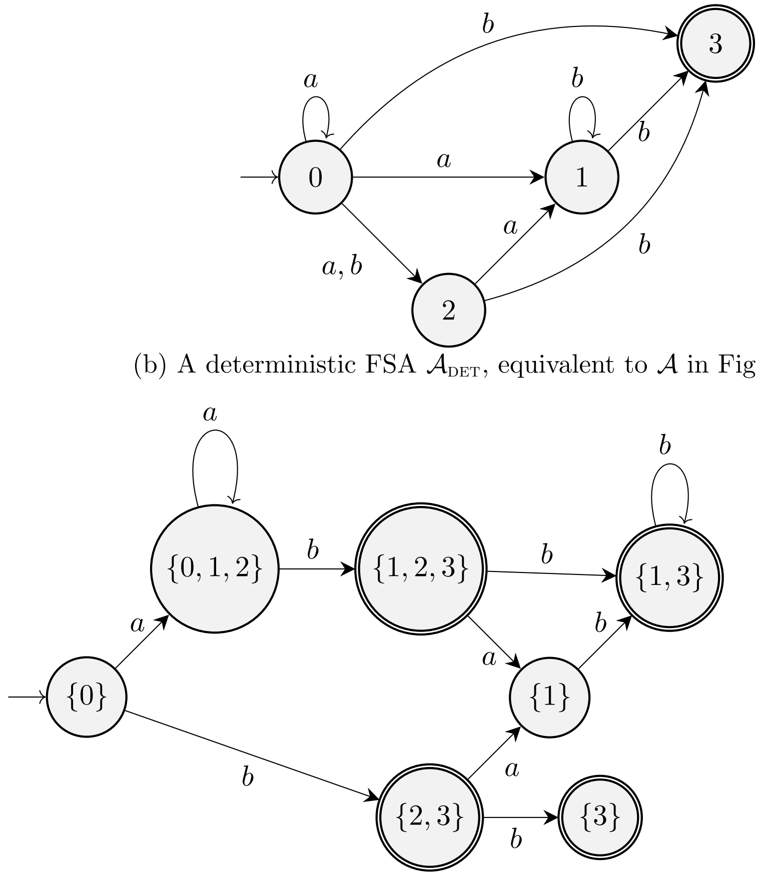
Example 3.10.1. *Let us now look at a non-deterministic FSA and its equivalent deterministic version. Consider the FSA in Fig. 3.25a. The determinized version is shown in Fig. 3.25b. As you can see, since the only starting state in $\mathcal{A}$ is 0, the initial state in $\mathcal{A}_{\text{DET}}$ is $\{0\}$. There are 3 states which can be reached from the initial state following an a-transition: 0 (self loop at 0), 1, and 2 (again, both from 0). These are therefore combined into the state $\{0, 1, 2\}$ and connected to the initial state via an a-transition. Furthermore, following a b-transition from any of the states 0, 1, or 2 can lead us into any of the states 1 (self loop at 1), 2 (from 0), or 3 (from 0 and 1). Therefore the state $\{1, 2, 3\}$ is initialized and connected via b to $\{0, 1, 2\}$. Since the set of reachable states following an a-transition from any of 0, 1, or 2 is again 0, 1, and 2, the same state gets created on Line 8 of the algorithm and a a-self loop gets added to $\mathcal{A}_{\text{DET}}$.*

To make the notation in the following sections more concise, we now also overload the definition of the transition function δ to strings and sets of states as follows.

Definition 3.10.1. *Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be an automaton, $q \in Q$, $\mathcal{Q} \subseteq Q$, and $\boldsymbol{y} \in \Sigma^*$. We define*

(i) $\delta(q, \boldsymbol{y})$ as the set of states reachable from q with a path with the yield $\boldsymbol{y}$*;*

(a) A FSA $\mathcal{A}$.



(b) A deterministic FSA $\mathcal{A}_{\text{DET}}$, equivalent to $\mathcal{A}$ in Fig. 3.25a.

Figure 3.25: An example of a non-deterministic FSA and its deterministic equivalent.

(ii) $\delta(\mathcal{Q}, \boldsymbol{y})$ *as the set of states* reachable from any $q \in \mathcal{Q}$ *with a path with the yield* $\boldsymbol{y}$*, i.e.,* $\delta(\mathcal{Q}, \boldsymbol{y}) = \bigcup_{q \in \mathcal{Q}} \delta(q, \boldsymbol{y})$.

We now prove the correctness of Determinize. To do so, we will make use of the following two lemmata.

Lemma 3.10.1. *Let* $\mathcal{A}_{\text{DET}} =$ Determinize$(\mathcal{A})$ *be the output of* Determinize *on the input* $\mathcal{A}$*. Then,* $\mathcal{A}_{\text{DET}}$ *is deterministic.*

Proof. First, we note that $\mathcal{A}_{\text{DET}}$ only has a single initial state. Second, we note that no ε-transitions are added to $\mathcal{A}_{\text{DET}}$. And, finally, by the if statement on Line 9, every power state $\mathcal{Q}$ is push at most 1. And, we can see from Line 8 at most one transition is added with $\mathcal{Q}$ as the source per symbol in the alphabet. This is the definition of determinism. ■

Lemma 3.10.2. *Let* $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ *be an* ε*-free FSA. Let* $\mathcal{A}_{\text{DET}} = (\Sigma, Q, I, F, \delta)_{\text{DET}} =$ Determinize$(\mathcal{A})$ *be the output of* Determinize *on the input* $\mathcal{A}$*. Then*

$$\delta_{\text{DET}}(I_{\text{DET}}, \boldsymbol{y}) = \bigcup_{q_I \in I} \delta(q_I, \boldsymbol{y}) \tag{3.172}$$

Proof. We prove the lemma by induction on the length of $\boldsymbol{y}$. For the base case, let $|\boldsymbol{y}| = 1$. Then, $\boldsymbol{y} = a \in \Sigma$. For any $a \in \boldsymbol{y}$, the state $\mathcal{Q}'$ reachable from $\mathcal{Q} = \{I\}$ is definitionally

$$\begin{aligned}\mathcal{Q}' &\leftarrow \left\{q : \bullet \xrightarrow{a} q \in \mathcal{E}\left(\{I\}\right)\right\} \\ &= \bigcup_{q_I \in I} \left\{q : \bullet \xrightarrow{a} q \in \mathcal{E}\left(q_I\right)\right\} \\ &= \bigcup_{q_I \in I} \delta\left(q_I, a\right)\end{aligned}$$

Now, by the inductive hypothesis, suppose that the hypothesis holds for any string $\boldsymbol{x}$ with $|\boldsymbol{x}| = n$. Let $\boldsymbol{y}$ be a string such that $|\boldsymbol{y}| = n + 1$. Then $\boldsymbol{y} = \boldsymbol{x}a$ where $|\boldsymbol{x}| = n$ and $a \in \Sigma$. Notice that $|\delta_{\text{DET}}\left(I_{\text{DET}}, \boldsymbol{y}\right)| = 1$ since $\mathcal{A}_{\text{DET}}$ is deterministic by Lemma 3.10.1. Therefore, we only have to look at *one* state $\mathcal{Q}'$. Let $\mathcal{Q} = \delta_{\text{DET}}\left(I_{\text{DET}}, \boldsymbol{x}\right)$. By our induction hypothesis, $\mathcal{Q} = \delta_{\text{DET}}\left(I_{\text{DET}}, \boldsymbol{x}\right) = \bigcup_{q_I \in I} \delta\left(q_I, \boldsymbol{x}\right)$. We now derive:

$$\begin{aligned}\mathcal{Q}' &\leftarrow \left\{q : \bullet \xrightarrow{a} q \in \mathcal{E}\left(\mathcal{Q}\right)\right\} && (3.173)\\ &= \bigcup_{p \in \mathcal{Q}} \delta\left(p, a\right) && (3.174)\\ &= \bigcup_{p \in \bigcup_{q_I \in I} \delta\left(q_I, \boldsymbol{x}\right)} \delta\left(p, a\right) && (3.175)\\ &= \bigcup_{q_I \in I} \bigcup_{p \in \delta\left(q_I, \boldsymbol{x}\right)} \delta\left(p, a\right) && (3.176)\\ &= \bigcup_{q_I \in I} \delta\left(q_I, \boldsymbol{x}a\right) && (3.177)\\ &= \bigcup_{q_I \in I} \delta\left(q_I, \boldsymbol{y}\right) && (3.178)\end{aligned}$$

This proves the result. ■

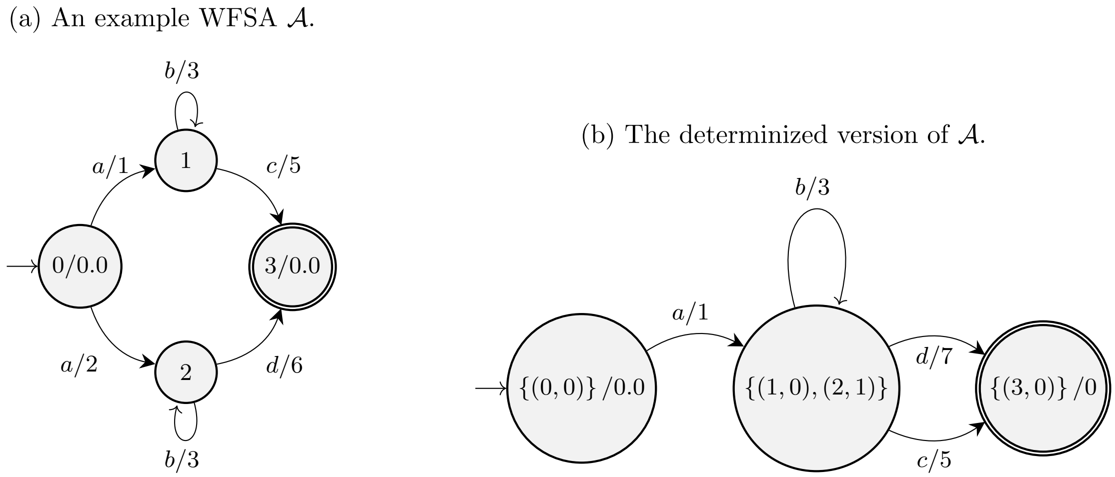
Theorem 3.10.1. *Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be an ε-free FSA. The output of $\mathcal{A}_{\text{DET}} = (\Sigma, Q, I, F, \delta)_{\text{DET}} =$ Determinize$(\mathcal{A})$ accepts the same language as $\mathcal{A}$.*

Proof. To prove the theorem, we have to show that $\forall\ \boldsymbol{y} \in \Sigma^*$, $\boldsymbol{y} \in L\left(\mathcal{A}\right) \iff \boldsymbol{y} \in L\left(\mathcal{A}_{\text{DET}}\right)$. By definition, a power state $\mathcal{Q} \in Q_{\text{DET}}$ is a final state if and only if there exists a $q \in \mathcal{Q}$ such that $q \in F$, i.e., when $\mathcal{Q} \cap F \neq \emptyset$. Suppose that $\boldsymbol{y} \in L\left(\mathcal{A}\right)$. Then, there exists a path from some state $q_I \in I$ to a state $q_F \in F$, yielding $\boldsymbol{y}$, i.e., $\delta\left(q_I, \boldsymbol{y}\right) \cap F \neq \emptyset$. By Lemma 3.10.2, $\delta_{\text{DET}}\left(I_{\text{DET}}, \boldsymbol{y}\right) = \mathcal{Q} = \bigcup_{q_I' \in I} \delta\left(q_I', \boldsymbol{y}\right)$. Since the union includes q_I, we have that $\delta\left(q_I, \boldsymbol{y}\right) \subseteq \mathcal{Q}$ and $\mathcal{Q} \cap F \neq \emptyset$, meaning that $\mathcal{Q}$ is final and $\boldsymbol{y} \in L\left(\mathcal{A}_{\text{DET}}\right)$. Now, suppose that $\boldsymbol{y} \in L\left(\mathcal{A}_{\text{DET}}\right)$. Let $\mathcal{Q} = \delta_{\text{DET}}\left(\boldsymbol{y}\right)$. Since $\mathcal{Q} \in F_{\text{DET}}$, $\mathcal{Q} \cap F \neq \emptyset$, i.e., there exists a final state $q_F \in \mathcal{Q} \cap F \neq \emptyset$ reachable from an initial state $q_I \in I$ with a path yielding $\boldsymbol{y}$. By Lemma 3.10.2, this implies $\bigcup_{q_I' \in I} \delta\left(q_I', \boldsymbol{y}\right) \cap F \neq \emptyset$, i.e, $\exists q_I \in I : \delta\left(q_I, \boldsymbol{y}\right) \cap F \neq \emptyset$. This is exactly the definition of acceptance by $\mathcal{A}$, meaning that $\boldsymbol{y} \in L\left(\mathcal{A}\right)$.

■

3.10.2 Weighted Determinization

Now we discuss how to generalize the unweighted determinization algorithm to the weighted case. As in the unweighted case, we will be working on constructing an *equivalent* deterministic automaton. A simple example of a non-deterministic but determinizable WFSA over the tropical semiring and an equivalent deterministic automaton is shown in Fig. 3.26.


(a) An example WFSA $\mathcal{A}$.

(b) The determinized version of $\mathcal{A}$.

Figure 3.26: An example of a determinizable WFSA over the tropical semiring and an equivalent deterministic WFSA from Allauzen and Mohri (2003). Notice that the states 1 and 2 are twins (defined below).

3.10.3 Preliminaries

The machinery we will develop in this section and the rest of the chapter will require us to work with a notion of multiplicative inverses. To avoid tedious details, we limit ourselves to *divisible* semirings in which those always exist and are unique. They are defined formally next. Note, however, that determinization can be defined more generally on *weakly left divisible semirings*, which are quickly introduced at the end of the section.

Definition 3.10.2 (Divisible Semirings or Semifields). *A semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is said to be* ***divisible*** *if the algebraic structure $(\mathbb{K}, \otimes)$ is a* ***group****, i.e., if all elements $x \in \mathbb{K} \setminus \{\mathbf{0}\}$ admit a $\otimes$-inverse x^{-1}. Another term for a divisible semiring is a* ***semifield****.*

We will also require that non-zero elements do not add up to $\mathbf{0}$, which is why we introduce **zero-sum-free** semirings.

Definition 3.10.3 (Zero-sum-free Semirings). *A semiring is* ***zero-sum-free*** *if $\forall x, y \in \mathbb{K}$:*

$$x \oplus y = \mathbf{0} \implies x = y = \mathbf{0}. \tag{3.179}$$

We additionally have to assume a small technical condition, which is that for any string $\boldsymbol{y} \in \Sigma^*$, the pathsum over all paths starting at an initial state and yielding $\boldsymbol{y}$ is non-$\mathbf{0}$: $w\left(P\left(I, \boldsymbol{y}, Q\right)\right) \neq \mathbf{0}$, or, equivalently, that $\boldsymbol{\alpha}\left(\boldsymbol{y}, q\right) \neq \mathbf{0}$ $\forall q \in Q$. This condition is always satisfied with *trim* weighted automata over any zero-sum-free semiring, since there all states are accessible, meaning that their $\boldsymbol{\alpha}$ values are non-$\mathbf{0}$, and these non-$\mathbf{0}$ values do not sum up to $\mathbf{0}$ (Mohri, 2009).

From this section on to the end of the chapter, we assume that all the semirings we are working in are both *divisible* as well as *commutative*. The weighted determinization algorithm will keep the gist of the power set construction around, i.e., it will consider power states that are elements of the powerset of states. However, we expand the notion of a power state to include weights. We will define a weighted power state.

Definition 3.10.4 (Weighted power states)**.** *A* ***weighted power state*** *$\mathcal{Q}$ is as a set of state–weight pairs, i.e., a subset of $Q \times \mathbb{K}$, or an element of $2^{Q\times\mathbb{K}}$. A weighted power state takes the form $\mathcal{Q} = \{(q_1, w_1), \ldots, (q_k, w_k)\}$.*

We will mostly refer to weighted power states simply as a **power states**. As should be obvious, if $\mathbb{K}$ is an infinite set, then the powerset $2^{Q\times\mathbb{K}}$ is an infinite set—this is an important contrast to the unweighted case. This observation is crucial as it leads to the unfortunate fact that not all WFSAs can be determinized and our determinization will not halt in general.

Now, consider a power state $\mathcal{Q} = \{(q_1, w_1), \ldots, (q_k, w_k)\}$. What does it mean to be in $\mathcal{Q}$? Similarly to the unweighted case, it means that we are simultaneously in the states $\{q_1, \ldots, q_k\}$. However, the interpretation of the weights is a bit more nuanced. Mohri (2009) refers to the weights $w_1, \ldots w_k$ as the **residual weights**. Their role will be described below. However, before we continue and formally introduce the residual weights, some clarification on the notation is necessary. In the unweighted case, we could directly make use of the definition of $\mathcal{E}$ from Chapter 1 on Line 7. For the weighted power states, we have to make a slight adjustment. Thus, we define $\mathcal{E}_Q\left(\{(q_1, w_1), \ldots, (q_k, w_k)\}\right) = \{q_1, \ldots, q_k\}$, which is essentially a projection back to the states.

Now we define the residual weights. Let $\mathcal{A}_{\text{DET}} = (\Sigma_{\text{DET}}, Q_{\text{DET}}, \delta_{\text{DET}}, \lambda_{\text{DET}}, \rho_{\text{DET}})$ be the automaton produced by the weighted determinization algorithm from the input automaton $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$. The residual weights are defined for *each* (elementary) state $q \in Q$ in some power state $\mathcal{Q} \in Q_{\text{DET}}$.

Definition 3.10.5 (Residual weight)**.** *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA over a semifield $\mathcal{W}$. We first define the* ***deterministic initial state*** *as*

$$\mathcal{Q}_I \overset{\text{def}}{=} \left\{(q_I, \lambda(q_I)) \mid q_I \in I\right\}. \tag{3.180}$$

We define the ***residual weights*** *(or just* ***residuals****) of states $q \in \mathcal{E}_Q(\mathcal{Q}_I)$ as*

$$r_{\mathcal{Q}_I}(q_I) \overset{\text{def}}{=} \lambda(q_I). \tag{3.181}$$

If $q \notin \mathcal{Q}_I$, then $r_{\mathcal{Q}_I}(q) \overset{\text{def}}{=} \boldsymbol{0}$. This is exactly the second component of each pair in Eq. (3.180).

Now, we recursively define the residual weights for non-initial power states. Let $\boldsymbol{y}b \in \Sigma^*$ be a string of length at least 1. Let $\mathcal{Q}'$ be the unique power state (set of states) reached by $\mathcal{A}$ after reading in $\boldsymbol{y}$ and $\mathcal{Q}$ the unique power state reached after reading the string $\boldsymbol{y}b$. The residual weight of the state q in the power state $\mathcal{Q}$ is recursively defined in terms of $\mathcal{Q}'$'s residuals as

$$r_{\mathcal{Q}}(q) \overset{\text{def}}{=} \bigoplus_{q' \in \mathcal{Q}'} \bigoplus_{q' \xrightarrow{b/w} q \in \mathcal{E}(q')} w'^{-1} \otimes r_{\mathcal{Q}'}(q') \otimes w \tag{3.182}$$

where we define

$$w' \overset{\text{def}}{=} \bigoplus_{q' \in \mathcal{Q}'} \bigoplus_{q' \xrightarrow{b/w} q \in \mathcal{E}(q')} r_{\mathcal{Q}'}(q') \otimes w \tag{3.183}$$

If $q \notin \mathcal{Q}$, then $r_{\mathcal{Q}}(q) \overset{\text{def}}{=} \boldsymbol{0}$.

Note that, in contrast to Eq. (3.182), the inner $\bigoplus$-sum in Eq. (3.183) is *not bound* to any $q \in Q$. Rather, it sums over all possible transitions from $q' \in \mathcal{Q}'$. The targets of the transitions in $\mathcal{E}(q')$ over all $q' \in \mathcal{Q}'$ form exactly the states in $\mathcal{Q}$. This is emphasized by coloring the bound variable in red, whereas all non-colored variables are unbound. The residual weights of a $q \in Q$ need not be the same across different power states that q belongs to.

Now we give a useful lemma that helps us interpret the residuals as the "percentage" of the forward weight $\boldsymbol{\alpha}_{\mathcal{Q}}$ that belongs to each $q \in \mathcal{Q}$ by showing that the residuals of every weighted power state sum to 1.

Lemma 3.10.3 (Residuals Sum to **1**)**.** *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA over a semifield $\mathcal{W}$ and $\mathcal{Q}$ a weighted power state reached by a string $\boldsymbol{y}b$ with $|\boldsymbol{y}b| \geq 1$ with the residuals as defined in Def. 3.10.5. Then,*

$$\bigoplus_{q \in \mathcal{Q}} r_{\mathcal{Q}}(q) = \mathbf{1} \tag{3.184}$$

Proof. The restriction that $|\boldsymbol{y}b| \geq 1$ means that there exists another power state $\mathcal{Q}'$ reached by $\boldsymbol{y}$ before arriving at $\mathcal{Q}$. Therefore, we have that

$$\bigoplus_{q \in \mathcal{Q}} r_{\mathcal{Q}}(q) = \bigoplus_{q' \in \mathcal{Q}'} \bigoplus_{q' \xrightarrow{a/w} q \in \mathcal{E}(q')} w'^{-1} \otimes r_{\mathcal{Q}'}\left(q'\right) \otimes w \qquad \text{(Def. 3.10.5)} \tag{3.185}$$

$$= w'^{-1} \otimes \bigoplus_{q' \in \mathcal{Q}'} \bigoplus_{q' \xrightarrow{a/w} q \in \mathcal{E}(q')} r_{\mathcal{Q}'}\left(q'\right) \otimes w \qquad \text{(distributivity)} \tag{3.186}$$

$$= w'^{-1} \otimes w' \qquad \text{(definition of } w'\text{)} \tag{3.187}$$

$$= \mathbf{1} \qquad (w'^{-1} \text{ is a weak inverse of } w') \tag{3.188}$$

■

This reaffirms the interpretation of the residuals. Each $q \in \mathcal{Q}$ caries a certain "fraction" of the weight w', meaning that they all sum to **1**. We further build on this interpretation in Lemma 3.10.5.

3.10.4 Mohri's Algorithm

Now that all the definitions are out of the way, we can finally present the weighted determinization algorithm, adapted from Mohri (1997), which is shown in Alg. 20. Before diving in formally, let us again try to understand what the algorithm is doing in words. States of the resulting determinized automaton $\mathcal{A}_{\text{DET}}$ are weighted power states. Let us try to build an intuition on what that means. A weighted power state $\mathcal{Q}$ in the determinized automaton $\mathcal{A}_{\text{DET}}$ is the unique state that is reached when starting in the initial power state by a path with the yield $\boldsymbol{y}$. We may identify $\mathcal{Q}$ with the set of pairs $(q, r) \in Q \times \mathbb{K}$ (q being a state in the original $\mathcal{A}$ and r the corresponding residual[17]) such that q can be reached from an initial state of the original automaton $\mathcal{A}$ by a path with *the same yield* $\boldsymbol{y}$; interestingly, we will prove in Lemma 3.10.5 that

$$\alpha_{\mathcal{A}}(\boldsymbol{y}, q) = \alpha_{\mathcal{A}_{\text{DET}}}(\boldsymbol{y}, \mathcal{Q}) \otimes r_{\mathcal{Q}}(q). \tag{3.189}$$

Note that, since $\mathcal{A}_{\text{DET}}$ is *deterministic*, the power state reached with a path with the yield $\boldsymbol{y}$ is *unique*. The notation $\alpha_{\mathcal{A}_{\text{DET}}}(\boldsymbol{y}, \mathcal{Q})$ is therefore somewhat redundant and we can write $\alpha_{\mathcal{A}_{\text{DET}}}(\boldsymbol{y})$. Notice that the *residual appears as the additional term on the right hand side of the expression.* This

[17] We drop the explicit reference to q and $\mathcal{Q}$ since they are clear from the context.

Algorithm 20 The WeightedDeterminize algorithm.

```
def WeightedDeterminize($\mathcal{A}$):
  $\mathcal{A}_{\text{DET}} \leftarrow (\Sigma, Q, \delta, \lambda, \rho)_{\text{DET}}$    ▷Initialize a new WFSA over the same semiring as $\mathcal{A}$.
  $\mathcal{Q}_I \leftarrow \{(q_I, \lambda(q_I)) \mid q_I \in I\}$
  $\lambda_{\text{DET}}(\mathcal{Q}_I) \leftarrow \mathbf{1}$
  stack $\leftarrow [\mathcal{Q}_I]$
  $Q_{\text{DET}} \leftarrow \{\mathcal{Q}_I\}$    ▷Add the initial state to $Q_{\text{DET}}$
  while |stack| $> 0$ :
    pop power state $\mathcal{Q}$ from stack
    if $\mathcal{E}_Q(\mathcal{Q}) \cap F \neq \emptyset$ :
      $F_{\text{DET}} \leftarrow F_{\text{DET}} \cup \{\mathcal{Q}\}$
      $\rho_{\text{DET}}(\mathcal{Q}) \leftarrow \bigoplus_{q \in \mathcal{Q}} r_{\mathcal{Q}}(q) \otimes \rho(q)$    ▷Note $\rho(q) = \mathbf{0}$ if $q \notin F$
    for $a \in$ symbols$(\mathcal{E}(\mathcal{E}_Q(\mathcal{Q})))$ :
      $w' \leftarrow \bigoplus_{p \in \mathcal{Q}} \bigoplus_{p \xrightarrow{a/w} q \in \mathcal{E}(p)} r_{\mathcal{Q}}(p) \otimes w$    ▷Calculate the weight of the new $a$-outgoing transition.
      $\mathcal{Q}' \leftarrow \left\{ \left( q, \bigoplus_{p \in \mathcal{Q}} \bigoplus_{p \xrightarrow{a/w} q \in \mathcal{E}(p)} w'^{-1} \otimes r_{\mathcal{Q}}(p) \otimes w \right) : \bullet \xrightarrow{a/\bullet} q \in \mathcal{E}(\mathcal{E}_Q(\mathcal{Q})) \right\}$    ▷New power state
      if $\mathcal{Q}' \notin Q_{\text{DET}}$ :
        $Q_{\text{DET}} \leftarrow Q_{\text{DET}} \cup \{\mathcal{Q}'\}$
        push $\mathcal{Q}'$ onto stack
      $\delta_{\text{DET}} \leftarrow \delta_{\text{DET}} \cup \left\{ \mathcal{Q} \xrightarrow{a/w'} \mathcal{Q}' \right\}$
  return $\mathcal{A}_{\text{DET}}$
```

motivates the term **residual weight** at state q for r—it is the unique additional weight assigned to $q \in Q(\mathcal{Q})$ that is not accounted for by the weight of the (unique) path up to the power state itself and is different for each state in $Q(\mathcal{Q})$. The reason we need to keep these residual weights around is to weight the incoming and the subsequent outgoing transitions from $\mathcal{Q}$ appropriately—not all states $q \in Q(\mathcal{Q})$ will have the same set of incoming and outgoing arcs (and they will be of course weighted differently). Therefore, when constructing transitions from $\mathcal{Q}$, the different states in $\mathcal{Q}$ need to contribute differently to their weights. A visualization of this is shown in Fig. 3.27.

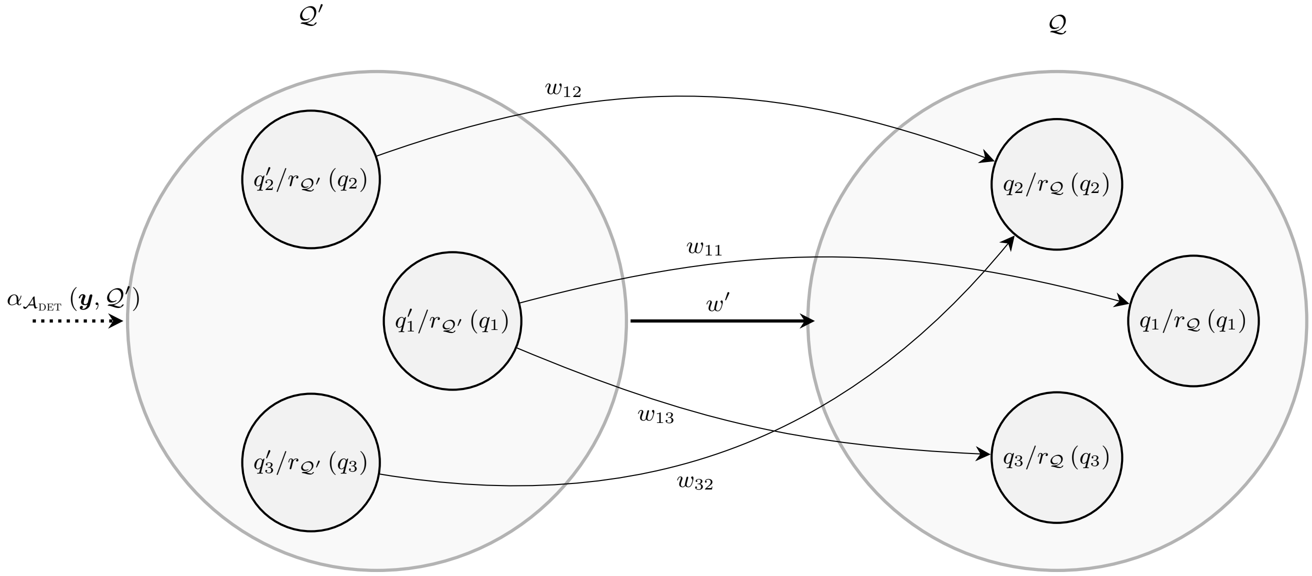

$\mathcal{Q}'$
$\mathcal{Q}$
w_{12}
$q_2'/r_{\mathcal{Q}'}(q_2)$
$q_2/r_{\mathcal{Q}}(q_2)$
w_{11}
$\alpha_{\mathcal{A}_{\text{DET}}}(\boldsymbol{y}, \mathcal{Q}')$
$q_1'/r_{\mathcal{Q}'}(q_1)$
w'
$q_1/r_{\mathcal{Q}}(q_1)$
w_{13}
$q_3'/r_{\mathcal{Q}'}(q_3)$
$q_3/r_{\mathcal{Q}}(q_3)$
w_{32}

Figure 3.27: A high-level view and intuition behind the residuals. For example, $\alpha_{\mathcal{A}}(\boldsymbol{y}, q_3') = \alpha_{\mathcal{A}_{\text{DET}}}(\boldsymbol{y}, \mathcal{Q}') \otimes r_{\mathcal{Q}'}(q_3')$ and $\alpha_{\mathcal{A}}(\boldsymbol{y}, q_3) = \alpha_{\mathcal{A}_{\text{DET}}}(\boldsymbol{y}, \mathcal{Q}') \otimes w' \otimes r_{\mathcal{Q}}(q_3') = \alpha_{\mathcal{A}_{\text{DET}}}(\boldsymbol{y}, \mathcal{Q}) \otimes r_{\mathcal{Q}}(q_3')$

Like Determinize, WeightedDeterminize uses a queue stack containing the set of states of the resulting automaton $\mathcal{A}_{\text{DET}}$ yet to be expanded. The queue discipline can be arbitrarily chosen and does not affect the termination of the algorithm. $\mathcal{A}_{\text{DET}}$ admits a unique initial state, ${q_I}'$, defined as the set of initial states of $\mathcal{A}$ augmented with their respective initial weights. Its initial weight is $\mathbf{1}$ (Lines 3–4). stack originally contains only the power state ${q_I}'$ (Line 4).

At each execution of the loop of Lines 6–17, a new power state p' is extracted from stack (Line 7). For each $a \in \Sigma$ labeling at least one of the transitions leaving a state p from the power state p', a new transition with input label a is constructed in the determinized automaton. The weight w' given to that transition is the $\oplus$-sum of the weights of all transitions in $\mathcal{E}(\mathcal{E}_Q(p'))$ which are labeled with a (therefore the transitions leading from a state p in the selected existing power state p' to a state $q \in q'$ in the newly created power state), pre-$\otimes$-multiplied by the *residual weight* $r_{p'}(p)$ at each state p (Line 9) (this state $p \in p'$ is the state in the original $\mathcal{A}$ carrying the particular outgoing transition). The *destination state* of the transition is the power state containing all the states q reached by transitions in $\mathcal{E}(\mathcal{E}_Q(p'))$ labeled with a. The residual of each state q of the power state q' is obtained by $\oplus$-summing over the transitions from the states $p \in p'$ into q where each term is obtained by post-$\otimes$-multiplying the residual weight of p by the weight of the transition from p leading to q and then left-dividing that by w' (remember, we can do that since we are in a *cancellative* semiring). Intuitively, the division ensures that the new transition weight corresponds to the "common factor" that all states q inside the newly created power state q' share, while the calculated residuals $r_{q'}(q)$ for $q \in q'$ correspond to the weight which needs to be post-$\otimes$-multiplied to it to get the individual state forward weights back.

The new subset q' is inserted in the queue stack when it is a new state (Line 15). If any of the states in the subset q' is final, q' is made a final state and its final weight is obtained by summing the final weights of all the final states in q', pre-$\otimes$-multiplied by their residual weight $r_{q'}(q)$ (Line 16). As before, the worst case complexity of WeightedDeterminize is exponential. However, in many practical cases such as for weighted automata used in large-vocabulary speech recognition, this blowup does not occur - the number of non-deterministic transitions in the original automaton is limited. This hopefully reveals the intuition behind weighted determinization. What is left is to show that WeightedDeterminize actually works correctly. To show correctness, we divide the proof into multiple parts. We start by showing two simple facts.

Proposition 3.10.1. *If the semiring $\mathcal{W}$ is* finite*, WeightedDeterminize always halts.*

Proof. We first make a simple observation: the size of any of the power sets WeightedDeterminize creates is never greater than $|Q|$, meaning that in any $\mathcal{Q} \in Q_{\text{DET}}$, each $q \in Q$ only appears at most once. That is because it is only added to the state with *one* weight—the $\oplus$-sum from Line 10, which is the residual as explained above. This means that the power states are all subsets of the set $Q \times \mathbb{K}$. Since both of the sets in the Cartesian product are finite by assumption, the number of its subsets is finite too. This means that the algorithm will only even produce (and expand) a finite number of weighted power states, resulting in eventual termination of the algorithm. ■

This means that an automaton over a finite semiring is always determinizable! It explains out weighted determinization fully generalizes the unweighted case as the Boolean semiring is finite.

Lemma 3.10.4. *Whenever the algorithm WeightedDeterminize halts on automaton $\mathcal{A}$, the resulting automaton $\mathcal{A}_{\text{DET}}$ is deterministic.*

Proof. The argument for $\mathcal{A}_{\text{DET}}$ being deterministic is identical to the one in Thm. 3.10.1. ■

Importantly, the statement of Lemma 3.10.4 is different than a proof of correctness. This lemma does not guarantee that the output preserves the semantics of the original machine $\mathcal{A}$, i.e., that it has the same strings. Next we prove one additional technical lemma before we proceed to proving the correctness of weighted determinization.

Lemma 3.10.5. *Let $\mathcal{A}$ be an ε-free WFSA over a semifield $\mathcal{W}$. Assume WeightedDeterminize halts on input $\mathcal{A}$ and let $\mathcal{A}_{\text{DET}} = (\Sigma, Q, \delta, \lambda, \rho)_{\text{DET}} =$ WeightedDeterminize$(\mathcal{A})$. Let $\boldsymbol{y} \in \Sigma^*$ and $\mathcal{Q} \in Q_{\text{DET}}$ the unique power state reached with the path yielding $\boldsymbol{y}$. Then, for every $q \in Q$, the following holds*

$$\boldsymbol{\alpha}_{\mathcal{A}}(\boldsymbol{y}, q) = \boldsymbol{\alpha}_{\mathcal{A}_{\text{DET}}}(\boldsymbol{y}) \otimes r_{\mathcal{Q}}(q) \tag{3.190}$$

where $r_{\mathcal{Q}}(q)$ is the residual of the state q in the power state $\mathcal{Q}$.

Proof. The proof proceeds by induction on the length of the string $\boldsymbol{y}$.

Base case. $\boldsymbol{y} = b \in \Sigma$ (—$\boldsymbol{y}| = 1$). Let $\mathcal{Q}$ be the unique power state in $\mathcal{A}_{\text{DET}}$ we arrive at after reading in $\boldsymbol{y} = b$ and let $\mathcal{Q}_I \xrightarrow{b/w'} \mathcal{Q}$. Then, by manipulation, we have

$$\begin{aligned}
\alpha_{\mathcal{A}}\left(\boldsymbol{y}, q'\right) &= \alpha_{\mathcal{A}}\left(b, q'\right) && \text{(definition)}\\
&= \bigoplus_{q_I \in I} \lambda\left(q_I\right) \otimes \bigoplus_{q_I \xrightarrow{b/w} q'} w && \text{(definition)}\\
&= \bigoplus_{q_I \in I} r_{\mathcal{Q}_I}\left(q_I\right) \otimes \bigoplus_{q_I \xrightarrow{b/w} q'} w && \text{(Line 3)}\\
&= w' \otimes w'^{-1} \otimes \bigoplus_{q_I \in I} r_{\mathcal{Q}_I}\left(q_I\right) \otimes \bigoplus_{q_I \xrightarrow{b/w} q'} w && (w' \otimes w'^{-1} = \mathbf{1})\\
&= w' \otimes \bigoplus_{q_I \in I} \bigoplus_{q_I \xrightarrow{b/w} q'} w'^{-1} \otimes r_{\mathcal{Q}_I}\left(q_I\right) \otimes w && \text{(distributivity)}\\
&= w' \otimes r_{\mathcal{Q}'}\left(q'\right) && \text{(Line 12)}\\
&= \alpha_{\mathcal{A}_{\text{DET}}}\left(b\right) \otimes r_{\mathcal{Q}'}\left(q'\right) && \text{(definition)}
\end{aligned}$$

Inductive step. Now, assume that the lemma holds for $|\boldsymbol{y}| < n$. Let $\boldsymbol{y} \in \Sigma^*$ be a prefix of length $n - 1$. Let $b \in \Sigma$ be an arbitrary letter and let $\mathcal{Q}$ be the unique power state in $\mathcal{A}_{\text{DET}}$ we arrive at after reading in $\boldsymbol{y}$. Again, basic manipulation shows the result:

$$\begin{aligned}
\boldsymbol{\alpha}_{\mathcal{A}}\left(\boldsymbol{y} \circ b, q'\right) &= \bigoplus_{q \in Q} \alpha_{\mathcal{A}}\left(\boldsymbol{y}, q\right) \otimes \bigoplus_{q \xrightarrow{b/w} q'} w && \text{(definition)}\\
&= \bigoplus_{q \in Q} \alpha_{\mathcal{A}_{\text{DET}}}\left(\boldsymbol{y}\right) \otimes r_{\mathcal{Q}}\left(q\right) \otimes \bigoplus_{q \xrightarrow{b/w} q'} w && \text{(inductive hypothesis)}\\
&= \alpha_{\mathcal{A}_{\text{DET}}}\left(\boldsymbol{y}\right) \otimes \bigoplus_{q \in Q} r_{\mathcal{Q}}\left(q\right) \otimes \bigoplus_{q \xrightarrow{b/w} q'} w && \text{(distributivity)}\\
&= \alpha_{\mathcal{A}_{\text{DET}}}\left(\boldsymbol{y}\right) \otimes w' \otimes w'^{-1} \otimes \bigoplus_{q \in Q} r_{\mathcal{Q}}\left(q\right) \otimes \bigoplus_{q \xrightarrow{b/w} q'} w && (w' \otimes w'^{-1} = \mathbf{1})\\
&= \left(\boldsymbol{\alpha}_{\mathcal{A}_{\text{DET}}}\left(\boldsymbol{y}\right) \otimes w'\right) \otimes \left(w'^{-1} \otimes \bigoplus_{q \in Q} r_{\mathcal{Q}}\left(q\right) \otimes \bigoplus_{q \xrightarrow{b/w} q'} w\right) && \text{(associativity)}\\
&= \left(\boldsymbol{\alpha}_{\mathcal{A}_{\text{DET}}}\left(\boldsymbol{y}\right) \otimes w'\right) \otimes \left(\bigoplus_{q \in Q} \bigoplus_{q \xrightarrow{b/w} q'} w'^{-1} \otimes r_{\mathcal{Q}}\left(q\right) \otimes w\right) && \text{(distributivity)}\\
&= \left(\boldsymbol{\alpha}_{\mathcal{A}_{\text{DET}}}\left(\boldsymbol{y}\right) \otimes w'\right) \otimes r_{\mathcal{Q}}\left(q'\right) && \text{(definition of } r\left(q'\right)\text{; Line 10)}\\
&= \boldsymbol{\alpha}_{\mathcal{A}_{\text{DET}}}\left(\boldsymbol{y} \circ b\right) \otimes r_{\mathcal{Q}}\left(q'\right) && \text{(definition of } \alpha_{\mathcal{A}_{\text{DET}}}\left(\boldsymbol{y} \circ b\right)\text{)}
\end{aligned}$$

This completes the proof. ■

The proof also shows why we need the technical condition that for any $\boldsymbol{y} \in \Sigma^*$, $\boldsymbol{\alpha}\left(\boldsymbol{y}, q\right) \neq \mathbf{0}\ \forall q \in Q$. It ensures that always $w' \neq \mathbf{0}$, i.e., that inverses w'^{-1} always exist. To see why, observe that if $\boldsymbol{\alpha}(\boldsymbol{y}, q)$ were $\mathbf{0}$ for all states in some equivalence class (which might only consist of a single state), the incoming edge w' into their power state would be $\mathbf{0}$, meaning that we would not be able to invert it.

Theorem 3.10.2. *Let $\mathcal{A}$ be an ε-free WFSA over a semifield $\mathcal{W}$. Let $\mathcal{A}_{\text{DET}}$ be the result of running* WeightedDeterminize *on $\mathcal{A}$. Then the $\mathcal{A}$ and $\mathcal{A}_{\text{DET}}$ have the same string weights for all strings in Σ^*.*

Proof. The proof follows a similar idea to the final part of the proof of correctness of the intersection algorithm. Let $\mathcal{Q}$ be the unique state we arrive at in $\mathcal{A}_{\text{DET}}$ when we read in $\boldsymbol{y}$.

$$\begin{aligned}
\mathcal{A}(\boldsymbol{y}) &= \bigoplus_{q\in Q} \boldsymbol{\alpha}_{\mathcal{A}}(\boldsymbol{y}, q) \otimes \rho(q) && \text{(definition)} && (3.191)\\
&= \bigoplus_{q\in Q} \boldsymbol{\alpha}_{\mathcal{A}_{\text{DET}}}(\boldsymbol{y}) \otimes r(q) \otimes \rho(q) && \text{(Lemma 3.10.5)} && (3.192)\\
&= \boldsymbol{\alpha}_{\mathcal{A}_{\text{DET}}}(\boldsymbol{y}) \otimes \left(\bigoplus_{q\in Q} r(q) \otimes \rho(q)\right) && \text{(distributivity)} && (3.193)\\
&= \boldsymbol{\alpha}_{\mathcal{A}_{\text{DET}}}(\boldsymbol{y}) \otimes \rho(\mathcal{Q}) && \text{(definition of } \mathcal{A}_{\text{DET}}\text{'s final weights)} && (3.194)\\
&= \mathcal{A}_{\text{DET}}(\boldsymbol{y}) && \text{(definition)} && (3.195)
\end{aligned}$$

This proves that WeightedDeterminize, if it halts, preserves the weight of all strings in Σ^*. ■

3.10.5 Appendix on Weighted Determinization: Weakly Divisible Semirings

As mentioned above, we can generalize the weighted determinization from divisible to *weakly left divisible* semiring. For completeness, this optional section defines them but spares the reader the details. Note that Mohri's algorithm still works with this more general type of semiring, and more on the topic can be found in Allauzen and Mohri (2003).

Definition 3.10.6 (Weakly left divisible Semirings)**.** *$\mathcal{W}$ is* ***left divisible*** *if every element $x \in \mathbb{K}\setminus\{\mathbf{0}\}$ admits a* left *inverse, i.e., $\forall x \in \mathbb{K}\setminus\{\mathbf{0}\}\ \exists y \in \mathbb{K} : y \otimes x = \mathbf{1}$ and $\mathcal{W}$ is said to be* ***weakly left divisible*** *if for any $x, y \in \mathbb{K}$ such that $x \oplus y \neq \mathbf{0}$, there exists* at least one *z such that $x = (x \oplus y) \otimes z$. The $\otimes$ operation is said to be* ***cancellative*** *if such a z is* unique. *If $\otimes$ is cancellative, the call the semiring cancellative as well.*

When z is not unique, we assume that we have an algorithm to find a canonical z and call it $z = (x \oplus y)^{-1} \otimes x$, i.e., we assume that z can be found in a consistent way, that is: $((u \otimes x) \oplus (u \otimes y))^{-1} \otimes (u \otimes x) = (x \oplus y)^{-1} x\ \forall x, y, u \in \mathbb{K},\ u \neq \mathbf{0}$.

Example 3.10.2 (Weakly left divisible semirings)**.** *The Tropical Semiring is weakly left divisible with $z = x - \min(x, y)$. This means that the $\otimes$ operation (addition) in the tropical semiring is also cancellative. Indeed, we see*

$$(x \oplus y) \otimes z = \min(x, y) + x - \min(x, y) = x \tag{3.196}$$

The probability semiring is also weakly left divisible with $z = \frac{x}{x+y}$, meaning the $\otimes$ operation is also cancellative. Again, we can confirm

$$(x \oplus y) \otimes z = (x + y) \cdot \frac{x}{x + y} = x \tag{3.197}$$

Both semirings are also zero-sum-free.

3.10.6 The Twins Property

We now move to the problem of determining *when a given automaton $\mathcal{A}$ can be determinized*. As we noted above, not all WFSA can be determinized. To see this, consider a very slightly modified automaton from Fig. 3.26. It is presented in Fig. 3.28a. It has the same topology, the only difference being the weights on the self-loops. Let us first inspect why the algorithm fails on this input.

3.10.7 How WeightedDeterminize Fails

The key thing to consider is that WeightedDeterminize constructs a new power state for each possible weighting of the states inside the power state. In the example from Fig. 3.26, the power state containing the states $\{1, 2\}$ is always reachable from itself. Therefore, whenever we expand $\{1, 2\}$ (with whatever residuals there may be), $\{1, 2\}$ is again a possible new reachable state. The algorithm will terminate only if at some point, the weights stop changing.

In the tropical semiring, the weight w' of the edge with the label b between the states $\{1, 2\}$ and $\{1, 2\}'$ is computed as

$$w' \stackrel{\text{def}}{=} \min_{\substack{p \in \{1,2\} \\ q \in p \in \{1,2\}'}} r(p) + w\left(p \xrightarrow{b/w} q\right). \tag{3.198}$$

The residuals start as 0 and 1. Then, after the first addition of the state $\{1, 2\}$, we get:

$$\begin{aligned} w' &\stackrel{\text{def}}{=} \min_{\substack{p \in \{1,2\} \\ q \in p \in \{1,2\}'}} r(p) + w\left(p \xrightarrow{b/w} q\right) && (3.199) \\ &= \min\left(r(1) + w\left(1 \xrightarrow{b/w} 1\right), r(1) + w\left(1 \xrightarrow{b/w} 2\right), r(2) + w\left(2 \xrightarrow{b/w} 1\right), r(2) + w\left(2 \xrightarrow{b/w} 2\right)\right) && (3.200) \\ &= \min(0 + 3, 0 + \infty, 1 + \infty, 1 + 4) = 3 && (3.201) \end{aligned}$$

Notice that this means that at any point of the algorithm, one of the states of the power state (in this case, state 1) will always have its residual be 0. This holds for the Tropical semiring in general, as we will see. This further implies that the weight w' will always be 3. The residuals of $q \in \{1, 2\}'$ are then calculated as

$$\min_{p \in \{1,2\}} r(p) + w\left(p \xrightarrow{b/w} q\right) - 3. \tag{3.202}$$

Particularly, for $q = 2$, we get

$$\begin{aligned} r(2') &= \min_{p \in \{1,2\}} r(p) + w\left(p \xrightarrow{b/w} 2\right) - 3 && (3.203) \\ &= \min\left(r(1) + w\left(1 \xrightarrow{b/w} 2\right) - 3, r(2) + w\left(1 \xrightarrow{b/w} 2\right) - 3\right) && (3.204) \\ &= r(2) + w\left(1 \xrightarrow{b/w} 2\right) - 3 && (3.205) \\ &= r(2) + 4 - 3 > r(2) && (3.206) \end{aligned}$$

This means that the residual weight for the state 3 will increase with every iteration. The algorithm thus produces an infinite number of states.

Intuitively, since the self-loop at 2 has a higher weight than that at 1, the residual at 2 will always have to "compensate" for that by being larger than 0. Moreover, since the weights add up as more self-loops at 1 and 2 are taken, and the weight of the path to 2 increases more every time, the residual has to increase more and more to still result in the same weight of the individual state 2.

3.10.8 Determining Determinizability

The problem with the determinizability of WFAS is actually two-fold. Firstly, there exist automata, which do not admit an equivalent subsequential automaton – these are called **non-subsequentiable**. Unfortunately, there also exist automata over some semirings which are not non-subsequentiable, but the determinization algorithm still does not halt on them. These are the so-called **non-determinizable** automata. If the determinization algorithm halts, given as input the automaton $\mathcal{A}$, we then say that $\mathcal{A}$ is **determinizable**. In general, it is not known if the sequentiability or the determinizability of an automaton $\mathcal{A}$ is decidable. Some results are available for specific semirings or classes of semirings, however. It, therefore, seems important to think of criteria that tell us when a WFSA is determinizable and when it is not, and then to design algorithms for testing determinizability. A simple criterion is acyclicity.

Theorem 3.10.3. *Acyclic WFSAs are determinizable.*

Proof. The number of distinct weighted power states created by Alg. 19 is upper-bounded by the number of unique strings labeling the paths in the original non-deterministic machine from an initial state (that is, each unweighted power state will result in at most as many weighted power sets as there are strings leading into that unweighted power state. Since this set of strings is final in an acyclic machine, the machine is determinizable. ■

In the rest of the section, we cover a more general and interesting criterion: the **twins property**.

We will see that the twins property characterizes relationships between states of an automaton $\mathcal{A}$. But for a pair of states to be twins, they obviously have to be **siblings** first. Note that below, we use the notation $w\left(P\left(q, \boldsymbol{y}, q\right)\right)$ as a shorthand for $\bigoplus_{\boldsymbol{\pi} \in P(q, \boldsymbol{y}, q)} w\left(\boldsymbol{\pi}\right)$. This is exactly what the algebraic paths algorithms we discussed in §3.6.1 compute (*without* the initial and final weights)!

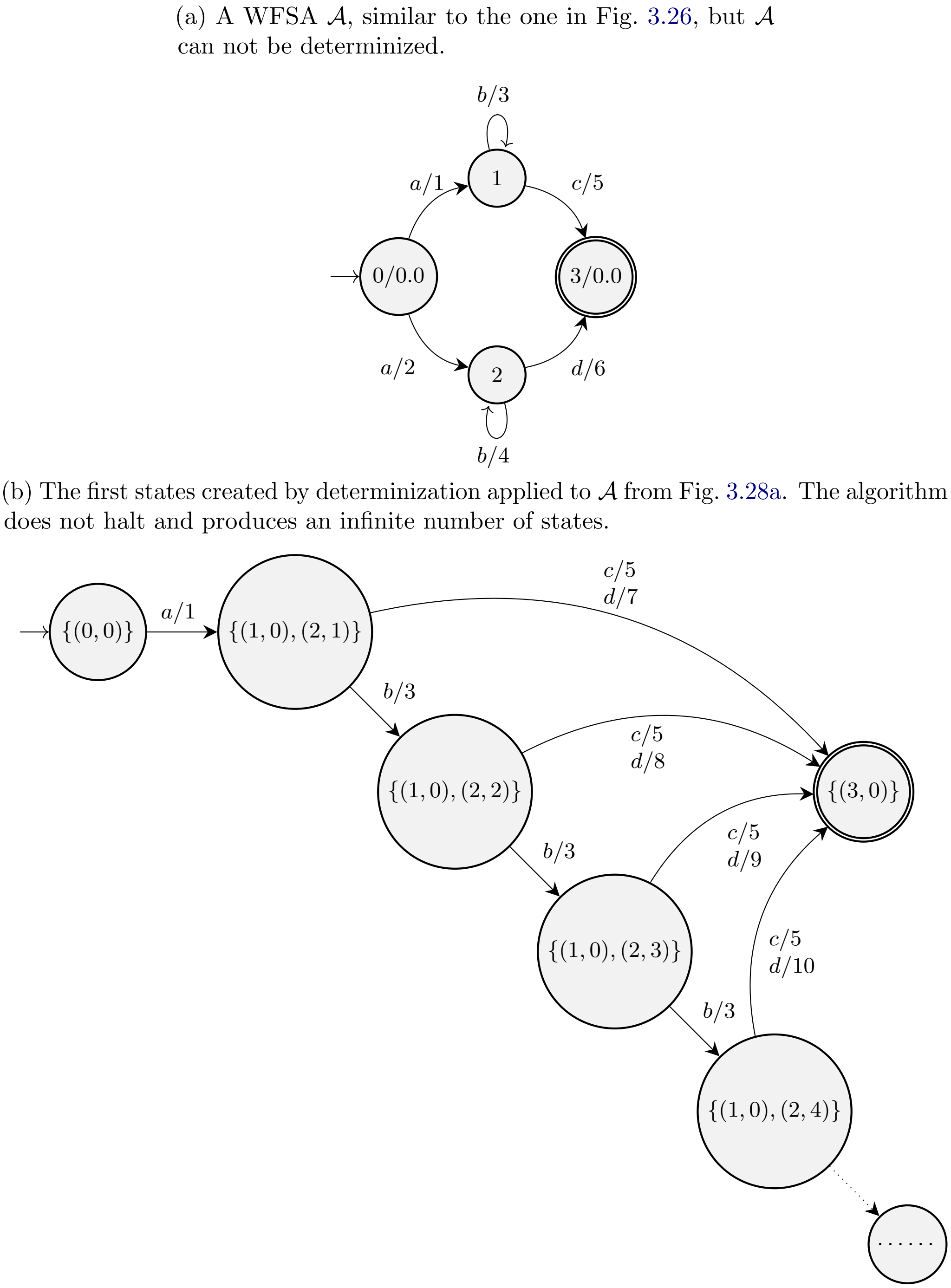
Definition 3.10.7 (Siblings)**.** *Let $\mathcal{A}$ be a weighted automaton over a weakly left divisible semiring $\mathcal{W}$. Two states q_1 and q_2 are said to be **siblings** if there exist two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ in Σ^* such that both q_1 and q_2 can be reached from I by paths labeled with $\boldsymbol{x}$ and there is a cycle at q_1 and a cycle at q_2 both labeled with $\boldsymbol{y}$.*

When $\mathcal{W}$ is both commutative *and* cancellative*, two sibling states are said to be twins if and only if for any string $\boldsymbol{y} \in \Sigma^*$:*

$$w\left(P\left(q_1, \boldsymbol{y}, q_1\right)\right) = w\left(P\left(q_2, \boldsymbol{y}, q_2\right)\right) \tag{3.207}$$

*$\mathcal{A}$ has the **twins property** if* any two sibling states of $\mathcal{A}$ are twins*.*

Note that there is a more general formulation of the twins property for weakly left divisible semirings, but the formulation in that case is way more elaborate—with commutative and cancellative semirings, it nicely simplifies to the form in Eq. (3.207). The expression Eq. (3.207) intuitively means that any cycles with the same yield starting and ending in sibling states must have the same weight. Notice that the states 1 and 2 in Fig. 3.26a are twins.


(a) A WFSA $\mathcal{A}$, similar to the one in Fig. 3.26, but $\mathcal{A}$ can not be determinized.

(b) The first states created by determinization applied to $\mathcal{A}$ from Fig. 3.28a. The algorithm does not halt and produces an infinite number of states.

Figure 3.28: An example of a non-determinizable WFSA with the same topology as the one in Fig. 3.26. Notice that the states 1 and 2 are siblings as before, but this time, they are *not twins*. Taken from Allauzen and Mohri (2003).

We will shortly see why the twins property could be useful, at least for the *tropical semiring*. Before we state the theorem, let us look at a binary operation over automata and a lemma that will help us prove the usefulness of the property.

Definition 3.10.8. *A* ***cross product*** *of two automata* $\mathcal{A}_1 = (\Sigma, Q, \delta, \lambda, \rho)_1$ *over the semiring* $\mathcal{W}_1$

and $\mathcal{A}_2 = (\Sigma, Q, \delta, \lambda, \rho)_2$ *over the semiring* $\mathcal{W}_2$ *where* $\Sigma_1 = \Sigma_2$ *is the automaton*

$$\mathcal{A}_1 \times \mathcal{A}_2 \overset{\text{def}}{=} (Q_1 \times Q_2, \Sigma_1, I_1 \times I_2, F_1 \times F_2, \delta, \lambda, \rho) \tag{3.208}$$

over the product semiring $\mathcal{W}_1 \times \mathcal{W}_2$ such that $(q_1, q_2) \xrightarrow{a/(w_1,w_2)} (q_1', q_2') \in \delta$ is a transition of $\mathcal{A}_1 \times \mathcal{A}_2$ if and only if $q_1 \xrightarrow{a/w_1} q_1' \in \delta_1$ and $q_2 \xrightarrow{a/w_2} q_2' \in \delta_2$. λ is defined as

$$\lambda\left((q_{I1}, q_{I2})\right) = \left(\lambda_1\left(q_{I1}\right), \lambda_2\left(q_{I2}\right)\right) \tag{3.209}$$

and ρ

$$\rho\left((q_{F1}, q_{F2})\right) = \left(\rho_1\left(q_{F1}\right), \rho_2\left(q_{F2}\right)\right). \tag{3.210}$$

Lemma 3.10.6. *Let* $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ *be a WFSA,* $\boldsymbol{y} \in \Sigma^*$ *and* $p, q, p', q' \in Q$. *Let also* $\boldsymbol{\pi} \in \Pi(p, \boldsymbol{y}, q)$ *and* $\boldsymbol{\pi}' \in \Pi(p', \boldsymbol{y}, q')$. *Assume* $|\boldsymbol{\pi}| > |Q|^2 - 1$ *and* $|\boldsymbol{\pi}'| > |Q|^2 - 1$. *Then, there exist strings* $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3 \in \Sigma^*$ *and states* p_1, p_2 *and* p_1', p_2' *such that* $|\boldsymbol{x}_2| > 0, \boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}_3 = \boldsymbol{y}$ *and such that* $\boldsymbol{\pi}$ *and* $\boldsymbol{\pi}'$ *can be* ***factored*** *in the following way:*

$$\begin{aligned}\boldsymbol{\pi} &\in \Pi\left(p, \boldsymbol{x}_1, p_1\right) \circ \Pi\left(p_1, \boldsymbol{x}_2, p_1\right) \circ \Pi\left(p_1, \boldsymbol{x}_3, q\right) \\ \boldsymbol{\pi}' &\in \Pi\left(p', \boldsymbol{x}_1, p_1'\right) \circ \Pi\left(p_1', \boldsymbol{x}_2, p_1'\right) \circ \Pi\left(p_1', \boldsymbol{x}_3, q'\right),\end{aligned}$$

where $\circ$ *here denotes the concatenation of sets, i.e.,*

$$S \circ T = \{st \mid s \in S, t \in T\}. \tag{3.211}$$

Proof. Consider the cross product of $\mathcal{A}$ with itself, $\mathcal{A} \times \mathcal{A}$. Let $\boldsymbol{\pi}$ and $\boldsymbol{\pi}'$ be two paths in $\mathcal{A}$ with lengths greater than $|Q|^2 - 1$ (i.e., $m > |Q|^2 - 1$:

$$\begin{aligned}\boldsymbol{\pi} &= \left((p = q_0, a_0, w_0, q_1), \ldots, (q_{m-1}, a_{m-1}, w_{m-1}, q_m = q)\right) \\ \boldsymbol{\pi}' &= \left((p' = q_0', a_0, w_0', q_1'), \ldots, (q_{m-1}', a_{m-1}, w_{m-1}', q_m' = q')\right).\end{aligned}$$

Then

$$\Pi = \left(\left(\left(q_0, q_0'\right), a_0, \left(w_0, w_0'\right), \left(q_1, q_1'\right)\right), \ldots, \left(\left(q_{m-1}, q_{m-1}'\right), a_{m-1}, \left(w_{m-1}, w_{m-1}'\right), \left(q_m, q_m'\right)\right)\right)$$

is a path in $\mathcal{A} \times \mathcal{A}$ with length greater than $|Q|^2 - 1$. Since $\mathcal{A} \times \mathcal{A}$ has exactly $|Q|^2$ states, Π admits at least one cycle at some state (p_1, p_1') labeled with a non-empty input string $\boldsymbol{x}_2$. ■

We now come to the central theorem of this section which characterizes determinizable automata over the tropical semiring.

Theorem 3.10.4. *Let* $\mathcal{A}$ *be a WFSA over the* tropical *semiring. If* $\mathcal{A}$ *has the twins property, then* $\mathcal{A}$ *is determinizable.*

Proof. First, let us write out the specific form the residuals take in the tropical semiring. Let $\mathcal{Q}'$ be a power state in $\mathcal{A}_{\text{DET}}$ returned by WeightedDeterminize and let $\mathcal{Q}$ be a state with an incoming transition into $\mathcal{Q}'$ labeled with b. We have

$$\begin{aligned} r_{\mathcal{Q}'}\left(q'\right) &\overset{\text{def}}{=} \left(\min_{q \in \mathcal{E}_Q(\mathcal{Q})} \min_{q \xrightarrow{b/w} q'} \left(r_{\mathcal{Q}}\left(q\right) + w\right)\right) - w' &(3.212)\\ &= \left(\min_{q \in \mathcal{E}_Q(\mathcal{Q})} \min_{q \xrightarrow{b/w} q'} \left(r_{\mathcal{Q}}\left(q\right) + w\right)\right) - \left(\min_{q'' \in \mathcal{Q}'} \min_{q \in \mathcal{E}_Q(\mathcal{Q})} \min_{q \xrightarrow{b/w} q''} \left(r_{\mathcal{Q}}\left(q\right) + w\right)\right). &(3.213)\end{aligned}$$

Notice that by definition of the min function, there exists in every power state $\mathcal{Q}$ a state q such that its residual is 0—the argmin of the second term on the second line of the equation.

Note that in the following, we use the alternative definition of the residuals $r_{\boldsymbol{y}}(q)$ to mean the $r_{\mathcal{Q}}(q)$ where $\mathcal{Q}$ is the power state arrived to with the string $\boldsymbol{y}$.

We prove the theorem by contradiction. Therefore, assume that $\mathcal{A}$ has the twins property. If the determinization algorithm does not halt,there exists at least one subset of Q, $\{q_0, \ldots, q_m\}$, such that the algorithm generates an infinite number of distinct weighted subsets, $\{(q_0, r_{\mathcal{Q}}(q_0)), \ldots, (q_m, r_{\mathcal{Q}}(q_m))\}$. Since the residual of at least one $q_i \in \mathcal{E}_Q(\mathcal{Q})$ will be 0, we have $m > 0$, i.e., $|\mathcal{Q}| > 0$. Let $S \subseteq \Sigma^*$ be the set of strings $\boldsymbol{y}$ for which $\{n(\boldsymbol{\pi}) : \boldsymbol{\pi} \in \Pi(I, \boldsymbol{y})\} = \{q_1, \ldots, q_m\}$, i.e., the strings which lead to *exactly* $\{q_1, \ldots, q_m\}$. S must be infinite, since otherwise, only a finite number of power states with the states $\{q_1, \ldots, q_m\}$ would be constructed—at most 1 per string.

Therefore, there exists a $i_0 \in [m]$ such that $r_{\boldsymbol{y}}(q_{i_0}) = 0$ for an infinite number of strings $\boldsymbol{y}$. Without loss of generality, assume $i_0 = 0$.

Let $T \subseteq S$ be the infinite set of strings $\boldsymbol{y}$ for which $r_{\boldsymbol{y}}(q_0) = 0$. Since the number of subsets $\{(q_0, 0), (q_1, r_{\boldsymbol{y}}(q_1)), \ldots, (q_m, r_{\boldsymbol{y}}(q_m))\}$ for $\boldsymbol{y} \in T$ is infinite, there exists a $j \in \{1, \ldots, m\}$ such that $r_{\boldsymbol{y}}(q_j)$ is distinct for an infinite number of strings $\boldsymbol{y} \in T$. Again, without loss of generality, we assume $j = 1$.

Now, let $U \subseteq T$ be an infinite set of string $\boldsymbol{y}$ with $r_{\boldsymbol{y}}(q_1)$ all distinct. Define $R(q_0, q_1)$ as the *finite* set of differences of the weights of paths leading to q_0 and q_1 labeled with the same string $\boldsymbol{y}$ with $|\boldsymbol{y}| \leq |Q|^2 - 1$:

$$\begin{aligned} R(q_0, q_1) \overset{\text{def}}{=} \{ & (\lambda(i_1) + w(\boldsymbol{\pi}_1)) - (\lambda(i_0) + w(\boldsymbol{\pi}_2)) : \\ & \boldsymbol{\pi}_0 \in \Pi(i_0, \boldsymbol{y}, q_0), \boldsymbol{\pi}_2 \in \Pi(i_1, \boldsymbol{y}, q_1), i_0, i_1 \in I, |\boldsymbol{y}| \leq |Q|^2 - 1\} \end{aligned}$$

The set is finite since we only consider strings of finite length from a finite alphabet. Moreover, it is non-empty since q_0 and q_1 are in the same power state and thus by definition reachable with the same strings.

Note that $\lambda(q) + w(q)$ denotes the forward weight of the state q and thus equals $w_{\mathcal{A}_{\text{DET}}}(\mathcal{Q}) + r_{\mathcal{Q}}(q)$. In the case of q_0 and $\boldsymbol{y}$, that means that $\lambda(q) + w(q) = w_{\mathcal{A}_{\text{DET}}}(\mathcal{Q}) + r_{\mathcal{Q}}(q) = w'$, i.e., it is equal to the incoming weight of the power state corresponding to $\boldsymbol{y}$.

We will show that $\{r_{\boldsymbol{y}}(q_1) \mid \boldsymbol{y} \in U\} \subseteq R(q_0, q_1)$, which will result in a contradiction with the infinity of U, proving that the algorithm terminates.

Let $\boldsymbol{y} \in U$. Consider the shortest path $\boldsymbol{\pi}_0 \in \Pi(i_0, \boldsymbol{y}, q_0)$ and the shortest path $\boldsymbol{\pi}_1 \in \Pi(i_1, \boldsymbol{y}, q_1)$ for i_0 and i_1 such that the quantities are minimal for $i \in I$. By definition of the subsets and the remark above, we have

$$(\lambda(i_1) + w(\boldsymbol{\pi}_1)) - (\lambda(i_0) + w(\boldsymbol{\pi}_2)) = (\lambda(i_1) + w(\boldsymbol{\pi}_1)) - w' = r_{\boldsymbol{y}}(i_1). \tag{3.214}$$

Suppose that $|\boldsymbol{y}| > |Q|^2 - 1$. By Lemma 3.10.6, there exists a factorization of $\boldsymbol{\pi}_0$ and $\boldsymbol{\pi}_1$ such that:

$$\begin{aligned} \boldsymbol{\pi}_0 &\in \Pi(i_0, \boldsymbol{x}_1, p_0) \circ \Pi(p_0, \boldsymbol{x}_2, p_0) \circ \Pi(p_0, \boldsymbol{x}_3, q_0) \\ \boldsymbol{\pi}_1 &\in \Pi(i_0', \boldsymbol{x}_1, p_0') \circ \Pi(p_0', \boldsymbol{x}_2, p_0') \circ \Pi(p_0', \boldsymbol{x}_3, q_0') \end{aligned}$$

with $|\boldsymbol{x}_2| > 0$. This, together with the assumed twins property, means that p_0 and p_1 are *twins*, i.e., $w(\Pi(p_0, \boldsymbol{y}, p_0)) = w(\Pi(p_1, \boldsymbol{y}, p_1))$. Next, we define $\boldsymbol{\pi}_0'$ and $\boldsymbol{\pi}_1'$ such that:

$$\begin{aligned} \boldsymbol{\pi}_0' &\in \Pi(i_0, \boldsymbol{x}_1, p_0) \circ \Pi(p_0, \boldsymbol{x}_3, q_0) \\ \boldsymbol{\pi}_1' &\in \Pi(i_0, \boldsymbol{x}_1, p_0) \circ \Pi(p_0, \boldsymbol{x}_3, q_0), \end{aligned}$$

i.e., paths with the yield such that the middle substring $\boldsymbol{x}_2$ is removed. Since $\boldsymbol{\pi}_0$ and $\boldsymbol{\pi}_1$ with the yield $\boldsymbol{y}$, $\boldsymbol{\pi}'_0$ and $\boldsymbol{\pi}'_1$ are the shortest paths with the yield $\boldsymbol{x}_1\boldsymbol{x}_3$. Therefore, we have $w(\boldsymbol{\pi}_0) = w(\boldsymbol{\pi}'_0) + w(\Pi(p_0, \boldsymbol{x}_2, p_0))$ and $w(\boldsymbol{\pi}_1) = w(\boldsymbol{\pi}'_1) + w(\Pi(p_1, \boldsymbol{x}_2, p_1))$.

This means that:

$$\begin{aligned}(\lambda(i_1) + w(\boldsymbol{\pi}_1)) - (\lambda(i_0) + w(\boldsymbol{\pi}_0)) &= \left(\lambda(i_1) + w\left(\boldsymbol{\pi}'_1\right) + w(\Pi(p_0, \boldsymbol{x}_2, p_0))\right) \\ &\quad - \left(\lambda(i_0) + w\left(\boldsymbol{\pi}'_0\right) + w(\Pi(p_1, \boldsymbol{x}_2, p_1))\right) \\ &= \left(\lambda(i_1) + w\left(\boldsymbol{\pi}'_1\right) + w(\Pi(p_0, \boldsymbol{x}_2, p_0))\right) \\ &\quad - \left(\lambda(i_0) + w\left(\boldsymbol{\pi}'_0\right) + w(\Pi(p_0, \boldsymbol{x}_2, p_0))\right) \quad \text{(twins property)} \\ &= \left(\lambda(i_1) + w\left(\boldsymbol{\pi}'_1\right)\right) - \left(\lambda(i_0) + w\left(\boldsymbol{\pi}'_0\right)\right) \\ &= r_{\boldsymbol{y}}(q_1)\end{aligned}$$

If $|\boldsymbol{x}_1\boldsymbol{x}_3| > |Q|^2 - 1$, we can repeat the factorization above again, this time with $\boldsymbol{y} = \boldsymbol{x}_1\boldsymbol{x}_2$. If, however, $|\boldsymbol{x}_1\boldsymbol{x}_3| \leq |Q|^2 - 1$, we have the shortest paths $\boldsymbol{\pi}'_0$ and $\boldsymbol{\pi}'_1$ from i_0 to q_0 and from i_1 to q_1 with lengths $\leq |Q|^2 - 1$ and such that $(\lambda(i_1) + w(\boldsymbol{\pi}'_1)) - (\lambda(i_0) + w(\boldsymbol{\pi}'_1)) = r_{\boldsymbol{y}}(q_1)$. Since $r_{\boldsymbol{y}}(q_1) = w(\boldsymbol{\pi}'_1) - w(\boldsymbol{\pi}'_0) \in R(q_0, q_1)$, $r_{\boldsymbol{y}}(q_1) \in R(q_0, q_1)$. Therefore, we have $\forall \boldsymbol{y} \in U, r_{\boldsymbol{y}}(q_1) \in R(q_0, q_1)$. However, $R(q_0, q_1)$ is *finite*, which means U *is finite as well.* This brings us to a contradiction, which finishes the proof. ■

The theorem motivates the need for an efficient algorithm for testing the twins property for weighted automata over the tropical semiring.

Unfortunately, the equivalent of Thm. 3.10.4 does not hold for the real or log semirings. Some infinitely ambiguous weighted automata over these semirings are not determinizable despite them having the twins property.

3.10.9 Testing the Twins Property

In this section, we describe a conceptually very simple algorithm for testing the twins property in commutative and cancellative semirings[18]. The algorithm comes from Allauzen and Mohri (2003).

By our definition of the twins property in for commutative and cancellative semirings, we have to test the property $w(P(q_1, \boldsymbol{y}, q_1)) = w(P(q_2, \boldsymbol{y}, q_2))\ \forall \boldsymbol{y} \in \Sigma^*$ for any two siblings q_1 and q_2.

Weighted Inverse As a part of the TestTwinsProperty algorithm, we will need to construct a **weighted inverse** of an automaton $\mathcal{A}$. A weighted inverse of the automaton $\mathcal{A}$ is the automaton $\mathcal{A}^{-1}$, in which the edge weights are the inverses of the the non-**0** weights of the original automaton:

$$q_1 \xrightarrow{a/w} q_2 \in \delta_{\mathcal{A}^{-1}} \iff q_1 \xrightarrow{a/w^{-1}} q_2 \in \delta_{\mathcal{A}}$$

Inverses of course do not exist in general semirings. However, in the case of cancellative semirings, they can always be *simulated*. The details are a bit technical, but you can find them in Allauzen and Mohri (2003). In the following, we just assume we can, for any $x \in \mathbb{K}$, find its (simulated) inverse x^{-1}.

The algorithm we develop below will require that, for any $q \in Q$, all the cycles starting and ending in q have unique yields. This characterizes **cycle-unambiguous** automata.

Definition 3.10.9 (Cycle-unambiguous Automata)**.** *An automaton $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ is **cycle-unambiguous** if for any state $q \in Q$ and any string $\boldsymbol{y}$ there is at most one cycle in q (i.e., starting and ending in q) labeled with $\boldsymbol{y}$.*

[18] Remember that whether this is "useful" when it comes to deciding determinizability does not matter.

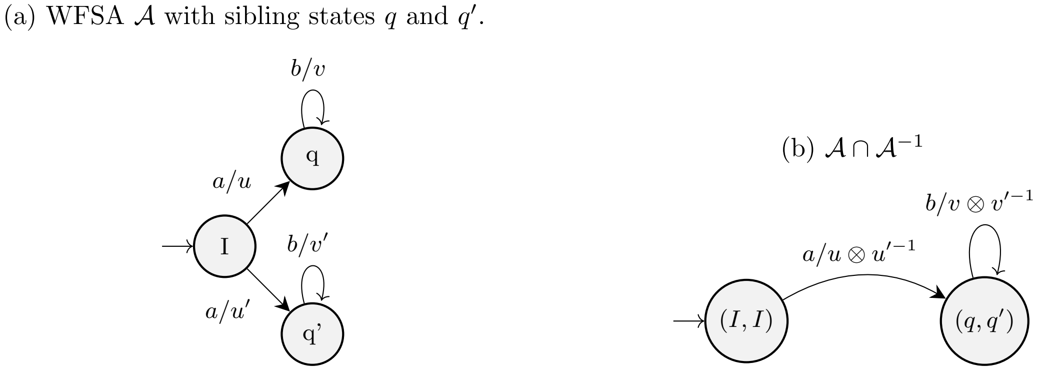
The following theorem is the core of the algorithm we will describe. It lays out a very simple criterion that needs to be tested for an automaton $\mathcal{A}$ to determine whether it has the twins property.

Theorem 3.10.5 (Twins Property)**.** *Let $\mathcal{A}$ be a trim cycle-unambiguous weighted automaton over $\mathcal{W}$. $\mathcal{A}$ has the twins property if and only if the* weight of any cycle *in $\mathcal{A} \cap \mathcal{A}^{-1}$ is $\mathbf{1}$.*

Proof. The proof follows Allauzen and Mohri (2003).

Let $\mathcal{A}$ be a trim cycle-unambiguous WFSA over $\mathcal{W}$. Since $\mathcal{A}$ is trim, the transition weights of $\mathcal{A}$ are all non-$\mathbf{0}$. Recall that the intersection algorithm combines state pairs if and only if they can be reached with the same input string $\boldsymbol{y}$. Therefore, two states q_1 and q_2 of $\mathcal{A}$ are siblings if and only if the state $q = (q_1, q_2)$ is constructed by the intersection algorithm on the input $\mathcal{A}$ and $\mathcal{A}^{-1}$ and there is a cycle c at q—the cycle required for q_1 and q_2 to be siblings. Let q_1 and q_2 be two such states, and let $\boldsymbol{y}$ be the label of c. By construction, the weight of c is the $\otimes$–product of the weights of a cycle c_1 in $\mathcal{A}$ at state q_1 labeled with $\boldsymbol{y}$ and the inverse of the weight of a cycle c_2 at q_2 (with the weights considered from the original automaton $\mathcal{A}$) labeled with $\boldsymbol{y}$: $w(c) = w(c_1) \otimes (c_2)^{-1}$. Since $\mathcal{A}$ is cycle-unambiguous, the cycles c_1 and c_2 are unique, and thus identical. Thus $\mathcal{A}$ has the twins property if and only if $w(c_1) = w(c_2)$, that is if and only if $w(c) = \mathbf{1}$. ■

Fig. 3.29 shows the graphical interpretation of the proof.


(a) WFSA $\mathcal{A}$ with sibling states q and q'.

(b) $\mathcal{A} \cap \mathcal{A}^{-1}$

Figure 3.29: The graphical illustration of the the proof of Thm. 3.10.5. The paths from I to q and I to q' with the same labels are matched by intersection. By Thm. 3.10.5, q and q' are twins if and only if $v = v'$, that is if and only if the weight $v \otimes v'^{-1}$ of the cycle in $\mathcal{A} \cap \mathcal{A}^{-1}$ is $\mathbf{1}$.

We can now finally introduce the algorithm for testing the twins property of an cycle-unambiguous automaton $\mathcal{A}$. Thm. 3.10.5 already heavily suggests how we should do that—we only need to check that the weights of any cycles in the automaton $\mathcal{A} \cap \mathcal{A}^{-1}$ equal $\mathbf{1}$. That can be done in linear time. Indeed, let S be a strongly connected component of $\mathcal{A} \cap \mathcal{A}^{-1}$ and q_S an arbitrary state of S. To test the desired property, we can run a depth-first search of S starting from q_S to compute the weight of any path from q_S to each state $q \in S$. That weight must be unique, otherwise there would be two cycles through q_S and q with distinct weights and the weight of one at least would be different from 1, as Fig. 3.30 illustrates.

Thus, we can define the value $d[q_S, q]$ as the weight of any path from q_S to q. The values $d[q_S, q]$ are initialized for some value UNDEFINED for $q \neq q_S$ and $\mathbf{1}$ for $q = q_S$. The pseudocode is presented in Alg. 21.

Lines 3–4 define $d[q_S, n(e)]$ as the weight of the first path from q_S to $n(e)$ found in a DFS of S. Lines 5–6 check that the weight of any other path from q_S to q found in a DFS of S equals $d[q_S, n(e)]$ and otherwise output FALSE. If this condition is never violated, the algorithm returns TRUE (Line 7).

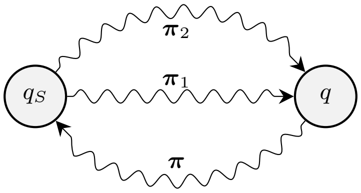


Figure 3.30: q_S and q are two states of the same strongly connected component of $\mathcal{A} \cap \mathcal{A}^{-1}$. If there are two paths $\boldsymbol{\pi}_1$ and $\boldsymbol{\pi}_2$ from q_S to q, then, since q_S and q are *in the same strongly connected component*, there is also a path $\boldsymbol{\pi}$ from q to q_S. $\boldsymbol{\pi}_1\boldsymbol{\pi}$ and $\boldsymbol{\pi}_2\boldsymbol{\pi}$ are cycles, thus $w(\boldsymbol{\pi}_1\boldsymbol{\pi}) = w(\boldsymbol{\pi}_2\boldsymbol{\pi}) = \mathbf{1}$ and $w(\boldsymbol{\pi}_1) = w(\boldsymbol{\pi}_2)$.

Algorithm 21 The CycleIdentityalgorithm.

```
def CycleIdentity($S$):
  for $e \in \mathcal{E}(S)$ :                ▷Edges are visited in the order of a DFS from $q_S$ in $S$.
    if $d[q_S, n(e)] =$ UNDEFINED :
      $d[q_S, n(e)] \leftarrow d[q_S, p(e)] \otimes w(e)$
    if $d[q_S, n(e)] \neq d[q_S, p(e)] \otimes w(e)$ :
      return FALSE
  return TRUE
```

To see that the algorithm does what we want, we have the following lemma.

Lemma 3.10.7. *The* CycleIdentity *algorithm returns* TRUE *if and only if the weight of any cycle in S equals $\mathbf{1}$.*

Proof. CycleIdentity returns TRUE if and only if for all transitions e in S, $d[q_S, n(e)] = d[q_S, p(e)] \otimes w(e)$.

By induction on the length of $\boldsymbol{\pi}$, this is equivalent to $d[q_S, n(\boldsymbol{\pi})] = d[q_S, p(\boldsymbol{\pi})] \otimes w(\boldsymbol{\pi})$ for any path $\boldsymbol{\pi}$ in S (left as an exercise). If $\boldsymbol{\pi}$ is a cycle, then $p(\boldsymbol{\pi}) = n(\boldsymbol{\pi})$, and thus $w(\boldsymbol{\pi}) = \mathbf{1}$. On the other hand, as mentioned above, if the weight of each cycle equals $\mathbf{1}$, all paths from q_S to any state q (including q_S) must have the same weight and thus the algorithm returns TRUE.

■

We can now wrap up the results of this section in the following theorem.

Theorem 3.10.6. *There exists an algorithm to test the twins property for any trim cycle-unambiguous weighted automaton $\mathcal{A}$ over $\mathcal{W}$ in time $\mathcal{O}\left(|Q|^2 + |\delta|^2\right)$.*

Proof. By Thm. 3.10.5, the test is equivalent to testing that the weight of any cycle of $\mathcal{A}' = \mathcal{A} \cap \mathcal{A}^{-1}$ equals $\mathbf{1}$. This can be done by running CycleIdentity for each strongly connected component S of $\mathcal{A}'$. The total cost of the algorithm is linear in the size of $\mathcal{A}'$ since a DFS can be done in linear time and since the strongly connected components of B can also be computed in linear time, as we saw in §3.6.7. Thus the complexity of the algorithm is $\mathcal{O}\left(|Q|^2 + |\delta|^2\right)$, which the time complexity of the intersection algorithm. ■

3.11 Minimization and Equivalence

This section deals with the question of testing whether two FSA are equivalent, which refers to the question of whether they accept the same weighted regular language. In order to do this, we will first introduce the concept of FSA equivalence more formally. We then look at different operations that maintain equivalence while transforming a WFSA into another WFSA that is canonical with respect to some property. The first such transformation is **weight pushing**, a technique that results in a canonicalization of the weights of a WFSA.The second is **minimization**, a technique that removes redundant states from a deterministic WFSA. The two techniques go hand in hand: Weight pushing was originally developed as a necessary first step to minimize a WFSA. However, pushing has also found its use cases when applied to non-deterministic WFSAs that we will also discuss. In contrast to the majority of the course, we will devote a large part of this chapter to developing algorithms for the minimization of *unweighted* automata. However, as we will see, this will lead to a very principled generalization to the weighted case. Finally, we look at different algorithms for testing the equivalence of weighted automata.

3.11.1 Unweighted Equivalence

We first have to inspect the important relation of *automata equivalence*, as it is the core principle of all approaches to minimization. We will start by looking at what it means for two *unweighted* FSA to be equivalent.

Our notion of equivalence of finite-state automata relies on the equivalence of their *states*. The equivalence of states, in turn, depends on whether two states have the same **right language**. A right language, which we define only intuitively here, is the language accepted if we treated the state as the only initial state in the FSA. Now we can formally definition of the equivalence of two states of an unweighted deterministic FSA:

Definition 3.11.1. *Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be an unweighted FSA. Two states $q, q' \in Q$ are* ***equivalent****, denoted with $q \sim_\delta q'$, if, for any string $\boldsymbol{y} \in \Sigma^*$, $\delta(q, \boldsymbol{y}) \cap F \neq \emptyset \iff \delta(q', \boldsymbol{y}) \cap F \neq \emptyset$. If two states are not equivalent, we say they are* ***distinguishable****.*

It is useful to think of the equivalence of states in an automaton as a mathematical relation between the states. Despite the "equivalence" in the name, the fact that the relation is an equivalence relation (from the perspective of the properties of mathematical relations) must be proved.

Theorem 3.11.1. *The equivalence of states q as defined in Def. 3.11.1 is an equivalence relation.*

Proof. We only prove that the relation is transitive. That is, in an automaton $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, if $q \sim_\delta q'$, and also $q' \sim_\delta q''$, then $q \sim_\delta q''$. The other two axioms of equivalence relations (reflexivity and symmetry) are easy to show and you are asked to formally prove them in exercise Exercise 3.18. Assume that $\sim_\delta$ is *not* transitive: suppose that $q \sim_\delta q'$ and $q' \sim_\delta q''$, while q and q'' are distinguishable. Then there exists a string $\boldsymbol{y}$ which distinguishes them, i.e., exactly one of $\delta(q, \boldsymbol{y})$ and $\delta(q'', \boldsymbol{y})$ is final. Suppose $\delta(q, \boldsymbol{y})$ is final (the alternative is symmetric). We now consider $\delta(q', \boldsymbol{y})$. If $\delta(q', \boldsymbol{y})$ is final, then q', q'' are distinguishable, since $\delta(q'', \boldsymbol{y})$ is not final. Similarly, if $\delta(q', \boldsymbol{y})$ is not final, q' and q'' are distinguishable. We conclude that q and q'' have to be equivalent. ■

In words, the equivalence of two states $q, q' \in Q$ implies that a string will end up in a final state starting in q if and only if it ends up in a final state starting in q'. Note that the actual final states do not have to be the same. Since $\sim_\delta$ is an equivalence relation, it defines a partition of the state space Q into equivalence classes.

3.11.2 Unweighted Minimization

We now start deriving the fundamental transformation of minimizing finite-state automata. We start by defining what a minimal automaton actually is.

Definition 3.11.2 (Minimal Automaton). *Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be an unweighted finite-state automaton. The automaton $\mathcal{A}'$ is a* ***minimal automaton*** *with regards to $\mathcal{A}$ if it is equivalent to $\mathcal{A}$ and is the automaton with the fewest states among all automata equivalent to $\mathcal{A}$.*

Minimization defines a transformation of an input *deterministic*[19] automaton into an automaton which has the fewest states among all *equivalent* automata. However, minimization does more than shrink the input automaton—it also serves as a *canonicalization* routine in that it transforms every deterministic automaton into a canonical form. Indeed, when coupled with Determinize, we can transform any determinizable FSA into a canonical form. This canonicalization then allows us to test whether any two FSAs represent the same language.

A closer look at state equivalence. The equivalence classes induced by the relation $\sim_\delta$ defined in Def. 3.11.1 lie at the heart of all minimization algorithms. Indeed, as we will see, minimization algorithms create a new FSA by *grouping* states of the original automaton into their equivalence classes and representing those classes as new states in the minimized machine. To see why this makes sense, consider a set of equivalent states $\mathcal{Q} = \{q_1, \ldots, q_k\}$ in some automaton $\mathcal{A}$. Suppose $\mathcal{A}$ ends up in some $q \in \mathcal{Q}$ upon reading some prefix of an input string $\boldsymbol{y}$. The equivalence of states in $\mathcal{Q}$ means that the remaining suffix of $\boldsymbol{y}$ will lead to a final state from some q if and only if it also leads to a final state from *any other* $q' \in \mathcal{Q}$. This means that we can *group* together the states in $\mathcal{Q}$ and replace the entire set of states with a *single* state. On the other hand, whenever two states are distinguishable, there must exist a string $\boldsymbol{y}$ which is in the right language or one of the states and not in the other's. We call such a string a **distinguishable extension**.

Example 3.11.1. *Let us look at the equivalence of states in an example. Consider the FSA in Fig. 3.31, taken from Hopcroft et al. (2006). The states C and G are not equivalent, since one is final and the other is not. Another way to say this is that the empty string ε distinguishes them. We will make use of this simple observation soon as we define our first minimization algorithm. Now consider the states A and G. The string 0 does not distinguish them (it leads to non-accepting states from both states), and the same holds for 1. However, the string 01 leads to C from A and to E from G. These target states are distinguished, meaning that the states A and G are as well. Again, the notion of "looking ahead" for distinguishing states will be a core idea of the algorithm presented shortly. Consider the states A and E, on the other hand. We can see that we transition to F upon reading 1 from any of the two states, meaning that no string beginning with 1 is able to distinguish them. Upon reading 0, however, we get to B from A and to H from E. This does not distinguish the states, since both targets are non-accepting. Furthermore, we end up in the same states upon reading either 0 or 1 at B and H—in C upon reading 1 and in G upon reading 0. This means that strings starting with 0 lead to the same state eventually as well and do not distinguish the A and E. Thus, the pair is equivalent.*

In the following, we will often use the set of **prefixes** of a language, defined as follows.

Definition 3.11.3 (Language Prefix). *Let L be a language. We define the set of* ***prefixes*** *of L as*

$$P_L = \{\boldsymbol{y} \in \Sigma^* \mid \exists \boldsymbol{x} \in \Sigma^* : \boldsymbol{y}\boldsymbol{x} \in L\} \tag{3.215}$$

[19] Recall that our definition of deterministic implies the automaton must be ε-free.

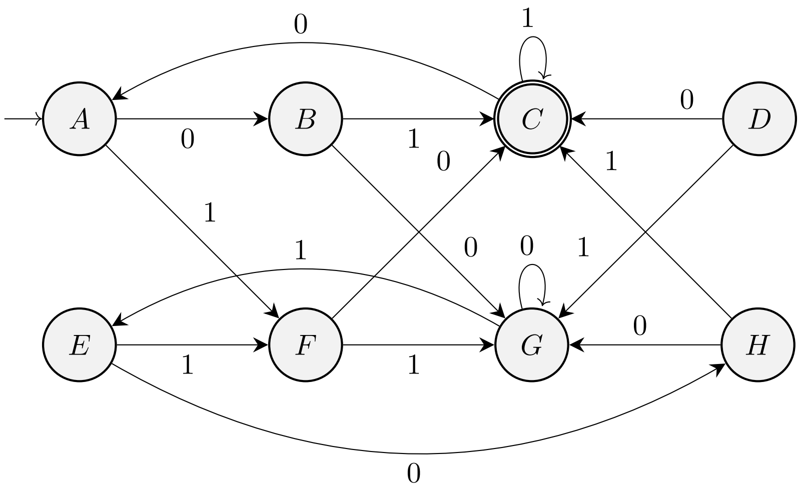


Figure 3.31: An example of a deterministic FSA.

The Myhill–Nerode Theorem

The Myhill–Nerode Theorem is a fundamental result in automata theory giving a necessary and sufficient condition for a language to be regular. It is closely related to the task of minimizing an automaton since it defines the minimal FSA accepting a given regular language. It makes use of the following equivalence relation—this time defined on a *language*.

Definition 3.11.4. *Let Σ be an alphabet, $L \subseteq \Sigma^*$ a language over Σ, and $\boldsymbol{x}, \boldsymbol{y} \in \Sigma^*$. We define a* ***distinguishing extension*** *to be a string $\boldsymbol{z} \in \Sigma^*$ such that* exactly one *of the two strings $\boldsymbol{xz}$ and $\boldsymbol{yz}$ belongs to L. We then define the relation $\sim_L$ on* all *strings over some alphabet as $\boldsymbol{x} \sim_L \boldsymbol{y}$ if and only if there is* no *distinguishing extension for $\boldsymbol{x}$ and $\boldsymbol{y}$. $\sim_L$ is called the* ***Nerode congruence****.*

Exercise Exercise 3.19 asks you to prove that $\sim_L$ is an equivalence relation. Nerode congruence is defined directly on the language itself, without the need to refer to any specific FSA accepting it. It is in fact very similar to the equivalence of states in an FSA through the equivalence of their languages. To make the important connection between the two equivalence relations, consider a DFA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ and its language $L = L(\mathcal{A})$. Since $\mathcal{A}$ is deterministic, any prefix of any string $\boldsymbol{y} \in L$ will lead to exactly one state in the automaton. Consider two prefixes $\boldsymbol{x}, \boldsymbol{y}$ and the states that they lead to: $q_{\boldsymbol{x}}$ and $q_{\boldsymbol{y}}$. By Def. 3.11.4, $\boldsymbol{x} \sim_L \boldsymbol{y}$ if there is no distinguishing extension of $\boldsymbol{x}$ and $\boldsymbol{y}$. What does $\boldsymbol{x} \sim_L \boldsymbol{y}$ mean for $q_{\boldsymbol{x}}$ and $q_{\boldsymbol{y}}$? Clearly, then any suffix starting from $q_{\boldsymbol{x}}$ will lead to a final state if and only if it leads to a final state from $q_{\boldsymbol{y}}$, since any suffix leading to a final state from only one of the two states would distinguish $\boldsymbol{x}$ and $\boldsymbol{y}$ in L. This means that $q_{\boldsymbol{x}} \sim_\delta q_{\boldsymbol{y}}$! It is not hard to see the converse holds as well—if $q_{\boldsymbol{x}} \sim_\delta q_{\boldsymbol{y}}$, it follows that $\boldsymbol{x} \sim_L \boldsymbol{y}$. This means that we can characterize the tight connection between the two relations as follows:

$$q_{\boldsymbol{x}} \sim_\delta q_{\boldsymbol{y}} \iff \boldsymbol{x} \sim_L \boldsymbol{y}. \tag{3.216}$$

Since $\sim_L$ is an equivalence relation, it partitions L into equivalence classes. These characterize the minimal deterministic FSA (DFA) accepting L, as the Myhill-Nerode Theorem states. For easier understanding, we will work up to it by first proving two related lemmata.

Lemma 3.11.1. *If a language L is regular, $\sim_L$ has a* finite number *of equivalence classes.*

Proof. Let L be a regular language. Then, by definition, there exists a DFA $\mathcal{A} = (\Sigma, Q, q_I, F, \delta)$ recognizing it. Start by partitioning L into at most $|Q|$ partitions, with the subset P_n consisting of the strings that, when input to $\mathcal{A}$, lead to the state q_n. For any two strings $\boldsymbol{x}, \boldsymbol{y} \in P_n$ and any possible extension $\boldsymbol{z}$, it holds that $n\left(\boldsymbol{\pi}_{\boldsymbol{xz}}\right) = n\left(\boldsymbol{\pi}_{\boldsymbol{yz}}\right)$, where $\boldsymbol{\pi}_{\boldsymbol{y}}$ refers to the unique accetping path of $\boldsymbol{y}$.[20] This means that $\mathcal{A}$ either accepts *both* $\boldsymbol{xz}$ and $\boldsymbol{yz}$ or *neither* of them, meaning that there is no distinguishing extension for $\boldsymbol{x}$ and $\boldsymbol{y}$, i.e., $\boldsymbol{x} \sim_L \boldsymbol{y}$. Each block P_n is therefore a contained in an equivalence class of $\sim_L$. On the other hand, every string $\boldsymbol{y} \in L$ is an element of some block P_n. This means that we have a many-to-one relation from Q to the equivalence classes of $\sim_L$, meaning that the number of the equivalence classes is bounded from above by $|Q|$. ■

Lemma 3.11.2. *Let L be a language. If $\sim_L$ has a* finite number *of equivalence classes, then L is regular.*

Proof. Assume $\sim_L$ has a finite number of equivalence classes for the language $L \subseteq \Sigma^*$. We denote the equivalence classes with $[\boldsymbol{y}]$, where $\boldsymbol{y} \in L$ is a string chosen to be a representative of the class.[21] We will show that there exists a DFA accepting L by constructing a DFA $\mathcal{A}_{\text{MIN}} = (\Sigma, Q_{\text{MIN}}, q_{I\text{MIN}}, F_{\text{MIN}}, \delta_{\text{MIN}})$ in the following manner. For every equivalence class $[\boldsymbol{y}]$, create a state $q_{[\boldsymbol{y}]}$. The initial q_I is the state $q_{[\varepsilon]}$ associated with the unique equivalence class of the string ε. For every pair of prefix strings $\boldsymbol{y}, \boldsymbol{y'}$ such that $\boldsymbol{y'} = \boldsymbol{y}a$ for some $a \in \Sigma$, we add a transition $q_{[\boldsymbol{y}]} \xrightarrow{a} q_{[\boldsymbol{y'}]}$ to δ. Finally, if the string $\boldsymbol{y}$ is in the actual language, i.e., $\boldsymbol{y} \in L$, we add $q_{[\boldsymbol{y}]}$ to the set of final states F.

■

Now we get to the main theorem. In fact, it turns out that the DFA we constructed in the proof of Lemma 3.11.2 is the *unique minimal* DFA accepting L.

Theorem 3.11.2 (Myhill–Nerode)**.** *Let L be a language such that $\sim_L$ has a finite number of equivalence classes. The DFA $\mathcal{A}_{\text{MIN}} = (\Sigma, Q_{\text{MIN}}, q_{I\text{MIN}}, F_{\text{MIN}}, \delta_{\text{MIN}})$ constructed as in the proof of Lemma 3.11.2 is the* minimal *DFA accepting L and it is also* unique *up to the renaming of the states.*

Proof. To prove that $\mathcal{A}_{\text{MIN}}$ is *minial*, suppose that there exists an equivalent automaton $\mathcal{A}' = (\Sigma, Q, q_I, F, \delta)'$ accepting L but with fewer states than $\mathcal{A}_{\text{MIN}}$. Let $\mathcal{U} = \mathcal{A}_{\text{MIN}} \cup \mathcal{A}'$. Note that we can construct the union without adding any new states. In the union, a state is final if and only if it is final in $\mathcal{A}_{\text{MIN}}$ or $\mathcal{A}'$. Any initial states in $\mathcal{A}_{\text{MIN}}$ and $\mathcal{A}'$ are equivalent, since the automata are assumed to be equivalent. Furthermore, for any pair of equivalent states q, q', their common descendants upon reading any input symbol a are also equivalent (otherwise q and q' would be distinguishable as well). $\mathcal{A}_{\text{MIN}}$ is trim by construction, and $\mathcal{A}'$ must also be trim, since otherwise we could trim them and get an even smaller equivalent automaton.This implies that every state $q \in Q_{\text{MIN}}$ is equivalent to *at least one* state $q' \in Q'$. We see this by the following. Let $q \in Q_{\text{MIN}}$. There is some string $\boldsymbol{y} = a_1 \ldots a_n$ which takes the start state q_I of $\mathcal{A}_{\text{MIN}}$ to the state q. It also takes the start state $q_I{}'$ of $\mathcal{A}'$ to some state q'. Since the two initial states are equivalent, the symbol a_1 takes them to equivalent states as well. From there, the same holds for a_2, then a_3, and so on, up to a_n. This means that q and q' are equivalent. By the pigeon-hole principle[22], since $\mathcal{A}'$ has fewer states than $\mathcal{A}_{\text{MIN}}$, there must be at least two states $q_1, q_2 \in Q_{\text{MIN}}$ which are equivalent to the

[20] Note that, since we are dealing with *deterministic* automata, any accepted string is the yield of exactly one path from an initial to a final state in the automaton.

[21] Since $\sim_L$ is an equivalence relation, we can use any string $\boldsymbol{y'} \in [\boldsymbol{y}]$ as the representative of $[\boldsymbol{y}]$.

[22] The pigeonhole principle states that when assigning m distinct items to n sets, with $m > n$, then at least one set will contain more than one item.

same state $q' \in Q'$. However, by transitivity, that means that the states q_1 and q_2 are equivalent themselves, which brings us to a contradiction. This means that there can be no automaton with fewer states than $\mathcal{A}_{\text{MIN}}$.

To prove *uniqueness*, we can reason in a similar way. Suppose $\mathcal{A}_1 = (\Sigma, Q, q_I, F, \delta)_1$ and $\mathcal{A}_2 = (\Sigma, Q, q_I, F, \delta)_2$ are two distinct minimal automata both accepting L. Again, we can argue that each state in $q_1 \in Q_1$ must be equivalent to at least one state $q_2 \in Q_2$. However, again, q_1 can not be equivalent to *more* than one state in Q_2 since we could then merge them. All states in Q_1 are therefore equivalent to exactly one state in Q_2, meaning that there is a one-to-one correspondence between them. Thus, the automata are equivalent up to the renaming of the states. ■

As mentioned above, the two equivalence relations described above play a key part when constructing minimal automata. In fact, all minimization algorithms we will see in the following sections work by finding the states that are equivalent under these equivalence relations and building a new FSA from those—just the part of finding the equivalence classes differs. The next section shows our first simple attempt at this, due to Moore.

Moore's Minimization Algorithm

We now introduce the first minimization algorithm for unweighted *deterministic* automata: Moore's algorithm. It closely follows the approach in which we found distinguishable states in Example 3.11.1, and this section describes Moore's algorithm almost as a direct translation of this reasoning. It is meant to introduce the task of minimization, as it proceeds in a similar way to other minimization algorithms—by first finding the equivalence classes of states under the the equivalence relation presented previously, and then grouping those classes into individual states. Later, we will develop a more general framework for this, called *partition refinement*, and see how to use it to improve this algorithm's efficiency.

The idea of the first minimization algorithm is very simple. It uses the following subroutine to find distinguishable pairs of states recursively:

- **Base Case**: For any pair of states q, q', if one of them is accepting and one is not, they are distinguishable.
- **Inductive Step**: Let q, q' be a pair of states. For some $a \in \Sigma$, let $p = \delta(q, a)$ and $p' = \delta(q', a)$. Now, we have the implication that if p and p' are distinguishable, then q and q' are distinguishable as well.

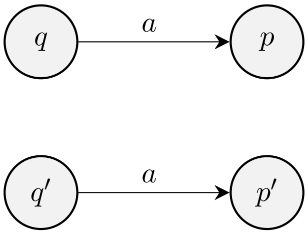


Figure 3.32: Inductive step. If p and q' are distinguishable, then so are q and q'.

We leave the implementation details to be discussed later in the chapter, and focus on more high-level ideas for the moment. The reasoning for the inductive step is as follows: To know that q and q' are distinguishable, we must find a string which distinguishes them. Since p and p' are distinguishable, we know that such a string $\boldsymbol{y}$ exists. Then, the string $a\boldsymbol{y}$ distinguishes q and q'.

The algorithm can easily be implemented by starting in pairs of states described in the base case and continuing the traversal backwards by exploring the states which can reach the states marked as distinguishable.

Example 3.11.2. *We demonstrate the process on the FSA from Example 3.11.1. Since C is the only final state, we start by marking all other states as distinguishable from C. We can keep track of distinguishable pairs in a table as shown in Tab. 3.1. We can then proceed onto the other state pairs and fill out the table sequentially. For instance, we see that, since C and H are distinguishable, E and F are as well—upon reading 0, they move to C and H, respectively. Indeed, any distinguishable pair except for A, G and E, G are discovered just by noticing one of them goes to C upon reading some symbol and the other does not. The remaining two distinguishable pairs are discovered in the subsequent round, since they end up in pairs discovered to be distinguishable in the previous round. We can also confirm that the states not distinguished by the table-filling algorithm are indeed equivalent. B, H and D, F will never be distinguished since both states in each pair lead to the same successors upon reading any input symbol. And finally, A, E go to F upon reading 1, and upon reading 0 they end up in B, H, an equivalent pair, meaning that A, E, too, are equivalent. The entire filled-in table is shown in Tab. 3.2.*

	A	B	C	D	E	F	G
B							
C		●					
D			●				
E			●				
F			●				
G			●				
H			●				

Table 3.1: The initialized table of Moore's algorithm. ● marks distinguishable pairs. Notice that only the lower triangular part of the table is shown since the relation is reflexive and symmetric.

	A	B	C	D	E	F	G
B	●						
C	●	●					
D	●	●	●				
E		●	●	●			
F	●	●	●		●		
G	●	●	●	●	●	●	
H	●		●	●	●	●	●

Table 3.2: The final table of produced in table-filling.

Due to the way the algorithm fills a table with information about distinguishable pairs, we will refer to this routine as table-filling. It turns out that this simple procedure finds, upon finishing, *all* possible equivalent pairs. This is summarized in the following theorem.

Theorem 3.11.3. *Two states are equivalent if and only if they are not distinguished by table-filling.*

Proof. ($\Longrightarrow$) We have to show that, if two states are marked as distinguished by table-filling, they are indeed distinguishable. We proceed with induction on the number of iterations (number of processed pairs) of table-filling:

- **Base Case**: *Initial iteration.* States marked as distinguished in the Base Case are clearly distinguishable, since they are distinguished by the empty string.

- **Inductive step**: *Iteration n.* Suppose that all states identified as distinguished in the previous $n-1$ iterations are distinguishable. Suppose the state pair q, q' is marked as distinguished in the n^{th} iteration of table-filling. Then, there is a symbol $a \in \Sigma$ which takes the pair q, q' to a pair p, p' previously marked as distinguished. But, by our inductive hypothesis, p, p' was marked as distinguished *correctly*, and can therefore be distinguished by some string $\boldsymbol{y} \in \Sigma^*$. Then, q, q' can be distinguished by the string $a\boldsymbol{y}$, meaning that q, q' are distinguishable as well.

($\Longleftarrow$) We prove this direction by contradiction. Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be the input automaton. Recall that we require $\mathcal{A}$ to be deterministic. Suppose that there exist some number M of pairs of states $(q_i, q_i'),\ i = 1, \ldots, M$ which are distinguishable, but table-filling does *not* distinguish them. We will call such pairs **bad pairs**. Let $\boldsymbol{y} = a_1 \cdots a_n$ be the *shortest* string which distinguishes a bad pair of states q, q'. Then exactly one of $\delta(q, \boldsymbol{y})$ and $\delta(q', \boldsymbol{y})$ is final. $\boldsymbol{y}$ cannot be ε, since that would imply one of the states q, q' is final and the other is not. This leads to a contradiction, since, in that case, q and q' would be distinguished in the base case of table-filling. This means that $n \geq 1$. Let $p = \delta(q, a_1)$ and $p' = \delta(q', a_1)$. Since $a_2 \cdots a_n$ distinguishes p and p', they are distinguishable as well. However, since the length of $a_2 \ldots a_n$ is shorter than that of $\boldsymbol{y}$, p, p' cannot be a bad pair by our assumption that $\boldsymbol{y}$ is the shortest string distinguishing any bad pair. This means that table-filling must have marked p, p' as distinguished. However, then, in the inductive part of the algorithm, the pair (q, q') would have been marked as distinguished as well, contradicting our assumption that it is a bad pair.

This finishes the proof. ■

Let us now see how table-filling can be used as a subroutine of a complete minimization algorithm—Moore's algorithm. The idea of what we have to do is simple. For each $q \in Q$, we have to construct a block consisting of q and all the states equivalent to it. This block is found using table-filling. This indeed follows a more general approach taken by other minimization algorithms as well. First, the states are partitioned into *blocks*, such that the states in the same block are *equivalent*, and no pair of states from different blocks are equivalent. These blocks then are converted into states of the equivalent output automaton. This construction is translated into the algorithm in Alg. 22.

Algorithm 22 The Moore algorithm.

```
def Moore($\mathcal{A} = (\Sigma, Q, q_I, F, \delta)$):
    $\mathcal{A}_{\text{MIN}} \leftarrow (\Sigma, Q_{\text{MIN}}, q_{I\text{MIN}}, F_{\text{MIN}}, \delta_{\text{MIN}})$    ▷Initialize the automaton $\mathcal{A}_{\text{MIN}}$ over same semiring as $\mathcal{A}$
    equivalent-pairs $\leftarrow$ table-filling($\mathcal{A}$)
    blocks $\leftarrow \{\}$    ▷Initialize empty map
    for $q \in Q$ :
        blocks[$q$] $\leftarrow$ equivalence-block(equivalent-pairs, $q$)    ▷Set blocks[$q$] to be the equivalence block of $q$.
        add blocks[$q$] to $Q_{\text{MIN}}$
    for $q \xrightarrow{a} q' \in \delta$ : add edge blocks[$q$] $\xrightarrow{a}$ blocks[$q'$]
    $q_{I\text{MIN}} \leftarrow$ blocks[$q_I$]
    for $q_F \in F$ : add blocks[$q_F$] to $F_{\text{MIN}}$
    return $\mathcal{A}_{\text{MIN}}$
```

Example 3.11.3. *Let us examine the algorithm in action by continuing our running example. We construct the minimized automaton of the FSA from Fig. 3.31.*

We start by constructing the minimized states by building block out of equivalent states found by the table filling algorithm. There is one starting state, A, E, since A is the only starting state in the original FSA[23]*. Analogously, the only final state is C. We see the edges have been added according to the presented algorithm—there is a directed edge with a label a whenever there is a transition in the original automaton with the same label between any of the states in the pair. For example, A, E goes to B, H upon reading 0 since A goes to B and E goes to H upon reading 0 in the original automaton. Similarly, since both A and E go to F upon reading 1, there is a 1-labeled edge between A, E and D, F. Notice that it does not matter that neither of A, E go to D—it is paired with F because they are equivalent.*

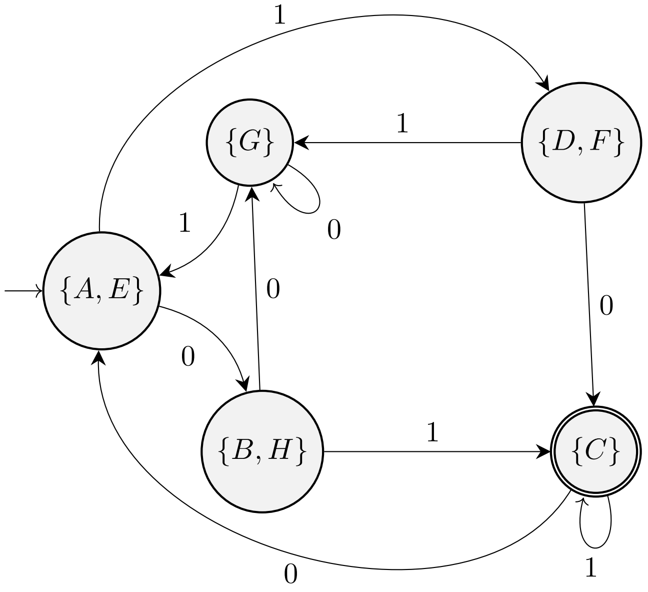


Figure 3.33: An example of a minimal FSA equivalent to the one in Fig. 3.31.

Minimality of the constructed FSA. The fact that table-filling finds equivalence classes of $\sim_\delta$, or, by Eq. (3.216), equivalently, the equivalence classes of $\sim_L$, allows us to conclude that Moore constructs the *unique minimal* equivalent automaton. This applies to any algorithm which constructs an automaton based on equivalence classes of $\sim_L$! Later, we will see a faster algorithm which works similarly—it only finds the equivalence classes in an asymptotically faster way. This concludes our introductory treatment of unweighted minimization. The next section introduces an abstract problem which will provide a common ground for our exploration of unweighted and weighted automata for the rest of the chapter.

[23] There is only one initial state because the FSA is deterministic.

3.11.3 Partition Refinement

We now look at the minimization of automata through a lens of a more generally applicable algorithm. This will allow us to present multiple minimization algorithms in a general framework, as is a common theme of this course. It turns out that, to understand the minimization of deterministic finite-state automata in general, it is actually easier to first engage with the more abstract problem of **partition refinement**. Thus, we first describe partition refinement in the abstract and then give an efficient algorithm for it. Afterwards, we will give a reduction from DFA minimization to partition refinement, viewing minimization as specific instances of this problem. We start by introducing partitions in general.

Definition 3.11.5 (Set Partition)**.** *Let U be a non-empty finite set. A **partition** $\mathcal{P}$ of U is a collection of subsets $\{P_1, \dots, P_N\}$ such that*

(i) they cover U: $U = \bigcup_{n=1}^{N} P_n$;

(ii) they are are pairwise disjoint, i.e., $P_n \cap P_{n'} = \emptyset$ for all $n, n' \in [N]$ where $n \neq n'$;

(iii) the empty set is not a member, i.e., $\emptyset \notin \mathcal{P}$.

*The elements P_n of $\mathcal{P}$ are called **blocks**.*

Example 3.11.4. *For example, let $U = \{1, 2, 3, 4, 5, 6\}$. Then a valid partition of U would be $\mathcal{P} = \{\{1\}, \{2, 3\}, \{4, 5, 6\}\}$. Another valid partition would be $\mathcal{Q} = \{\{1\}, \{2, 3\}, \{4\}, \{5, 6\}\}$.*

The high-level goal of partition refinement is to keep *refining* an initial partition into a more "fine-grained" one until it satisfies some specified criterion of "granularity". Importantly, the refined partitions still have to be "compatible" with the initial one. Let us now look at making the scare-quoted terms more precise.

Definition 3.11.6 (Partition Refinement)**.** *Let $\mathcal{P}$ and $\mathcal{Q}$ be two partitions of U. We say $\mathcal{Q}$ is a **refinement** of $\mathcal{P}$, denoted $\mathcal{Q} \preccurlyeq \mathcal{P}$, if, for every block $P \in \mathcal{P}$, there exists a subset of blocks $\widetilde{\mathcal{Q}} \subseteq \mathcal{Q}$ such that $P = \bigcup_{Q \in \widetilde{\mathcal{Q}}} Q$. Furthermore, if $\mathcal{Q}$ is a refinement of $\mathcal{P}$, we say $\mathcal{P}$ is **coarser** than $\mathcal{Q}$ and that $\mathcal{Q}$ is **finer** than $\mathcal{P}$.*

Intuitively, this means that if $\mathcal{Q} \preccurlyeq \mathcal{P}$, $\mathcal{Q}$ is more "fine-grained" than $\mathcal{P}$. $\mathcal{Q}$'s blocks are "smaller" and they further partition the blocks of $\mathcal{P}$. Importantly, since any block in $\mathcal{P}$ is *exactly* the union of a certain number of blocks in $\mathcal{Q}$, this means that no block in $\mathcal{Q}$ contains elements of more than 1 block in $\mathcal{P}$. In the example above, the partition $\mathcal{P}$ is clearly coarser than $\mathcal{Q}$, as $\mathcal{Q}$ has divided $\{4, 5, 6\}$ into the more fine-grained $\{4\}, \{5, 6\}$ (and is otherwise identical).

With that, we can introduce the criterion we will try to satisfy when modifying a partition. It is the notion of **stability** of a partition with regards to some function on the original set.

Definition 3.11.7 (Stability)**.** *Let U be a finite set and $f : U \to U$ a total function on U. Consider subset $B, S \subseteq U$.*

- *We say that B is **stable** with respect to S if either $B \subseteq f^{-1}(S)$ or $B \cap f^{-1}(S) = \emptyset$.*
- *We say that a partition $\mathcal{P}$ of U is **stable** with respect to S if all the blocks in $\mathcal{P}$ are stable with respect to S.*
- *We say that the partition $\mathcal{P}$ is **stable** if it is stable with respect to each of its own blocks.*

Another way of phrasing the last point is that for any $P \in \mathcal{P}$ and $x, y \in P$, $f(x)$ and $f(y)$ land in the same block P of $\mathcal{P}$.

Definition 3.11.8 (Stable Refinement)**.** *Let $\mathcal{P}$ be a partition of U and let $f : U \to U$ be a total function on U. A partition $\mathcal{R}$ is a **stable refinement** of $\mathcal{P}$, denoted $\mathcal{R} \preccurlyeq_f \mathcal{P}$, if $\mathcal{R}$ is stable with respect to f and $\mathcal{R} \preccurlyeq \mathcal{P}$. Furthermore, $\mathcal{R}$ is the **coarsest stable refinement**, denoted $\mathcal{R} \preccurlyeq_f^+ \mathcal{P}$, if $\mathcal{R} \preccurlyeq_f \mathcal{P}$, and, for any other stable refinement $\mathcal{R}' \preccurlyeq_f \mathcal{P}$, we have $\mathcal{R}' \preccurlyeq \mathcal{R}$.*

The Partition Refinement Problem. Whew! With the definitions out of the way, we are able to state the problem we seek to solve: Given an input partition $\mathcal{P}$ of U and a total function $f : U \to U$, we seek to find the *coarsest stable refinement*. That is, we are seeking a partition of U which is a *refinement* of $\mathcal{P}$ (it is in that sense "compatible" with the original partition) and is additionally *stable* with regards to f (it is also "compatible" with the function f). The input partition $\mathcal{P}$ is of course generally not stable with regards to f. With the problem defined, we will now develop an efficient algorithm for finding the coarsest partition refinement, due to Hopcroft (1971), called Hopcroft. It runs in time $\mathcal{O}(N \log N)$, where $|U| = N$. A major part of the algorithm revolves around *splitting* the existing blocks of a partition with regards to *preimages* of f until the partition is stable.

Definition 3.11.9 (Preimage)**.** *Let $f : U \to U$ be some function and $S \subseteq U$ some set. We define the **preimage** of S as $f^{-1}(S) \stackrel{\text{def}}{=} \{u \mid f(u) \in S\}$.*

Hopcroft partitions with respect to some block P_i those blocks P_j which map at least one element to P_i and one to $U \setminus P_i$. To formalize this, we introduce **splitting**.

Definition 3.11.10. *Let $S, P \subseteq U$. We define the function $\mathsf{split}(S, P)$, which is to be read as splitting P with S, as follows:*

$$\mathsf{split}(S,P) \stackrel{\text{def}}{=} \begin{cases} \{P \cap S, P \setminus S\} & \textbf{\textit{if }} P \cap S \neq \emptyset \textbf{\textit{ and }} P \setminus S \neq \emptyset \\ \{P\} & \textbf{\textit{otherwise}} \end{cases} \tag{3.217}$$

We additionally overload the function split to take a partition as its second argument

$$\mathsf{split}(S,\mathcal{P}) \stackrel{\text{def}}{=} \bigcup_{P \in \mathcal{P}} \mathsf{split}(S,P). \tag{3.218}$$

*The set S used for splitting is called a **splitter**.*

Given a splitter B, split therefore splits a block P_j into $\{u \mid u \in P_j, f(u) \in B\}$ and $\{u \mid u \in P_j, f(u) \notin B\}$. We also have the following equivalent characterization of a stable partition. A partition $\mathcal{P}$ is stable if, for all P in $\mathcal{P}$, $\mathcal{P} = \mathsf{split}(f^{-1}(P), \mathcal{P})$.

Lemma 3.11.3. *Let $\mathcal{P}$ be a partition of U and let S be a subset of U. Then, $\mathsf{split}(S, \mathcal{P})$ is also a partition of U, and, moreover, $\mathsf{split}(S, \mathcal{P}) \preccurlyeq \mathcal{P}$.*

Proof. We will show that the $\mathsf{split}(S, \mathcal{P})$ satisfies all properties required for a partition. The empty set is clearly not an element of $\mathsf{split}(S, \mathcal{P})$, since the sets in both cases in Eq. (3.217) contain at least one element and all block $P \in \mathcal{P}$ are non-empty. We now show that for any $x \in U$, x is an element of exactly one block in $\mathsf{split}(S, \mathcal{P})$, which is equivalent to $\mathsf{split}(S, \mathcal{P})$ covering U and having only disjoint blocks. Since $\mathcal{P}$ is a partition, $x \in P$ for some $P \in \mathcal{P}$. P either gets split into $P \cap S$ and $P \setminus S$ or remains unaltered. In both cases, the x ends up in *exactly* one block of $\mathsf{split}(S, \mathcal{P})$ by definition of the intersection and set difference operations. To see that $\mathsf{split}(S, \mathcal{P}) \preccurlyeq \mathcal{P}$, note that, P is the union of either one or two blocks in $\mathsf{split}(S, \mathcal{P})$ by the definition of split. ■

Lastly, we will make use of the following basic properties of splitting when developing partition refinement algorithms. The proofs are simple and involve some elementary set algebra. You are asked to formally prove them in the exercises.

Lemma 3.11.4 (Inheritance of Stability under Refinement). *Let $\mathcal{P}$ be a partition of U, let $\mathcal{R} \preccurlyeq \mathcal{P}$, and let $S \subseteq U$. Then, if $\mathcal{P}$ is stable with respect to S, then so is $\mathcal{R}$.*

Lemma 3.11.5 (Inheritance of Stability under Union). *Let $\mathcal{P}$ be a partition of U, and let $S', S'' \subseteq U$. Then, if $\mathcal{P}$ is stable with respect to S' and S'', then $\mathcal{P}$ is stable with respect to $S = S' \cup S''$.*

Lemma 3.11.6 (Monotonicity). *Let $\mathcal{P}$ be a partition of U, let $\mathcal{R} \preccurlyeq \mathcal{P}$, and let $S \subseteq U$. Then, $\mathit{split}(f^{-1}(S), \mathcal{R}) \preccurlyeq \mathit{split}(f^{-1}(S), \mathcal{P})$.*

Lemma 3.11.7 (Commutativity). $\mathit{split}(A, \mathit{split}(B, \mathcal{P})) = \mathit{split}(B, \mathit{split}(A, \mathcal{P}))$.

A First Attempt at Partition Refinement

We now describe our first attempt at an algorithm for partition refinement. We start with a simple approach by maintaining a partition $\mathcal{Q}$, which initially equals $\mathcal{P}$, and refining it until it is the coarsest stable refinement of $\mathcal{P}$. The algorithm repeats the following step until $\mathcal{Q}$ is stable:

Refine: Choose a block $S \in \mathcal{Q}$ such that it splits $\mathcal{Q}$. Replace $\mathcal{Q}$ with $\mathsf{split}(S, \mathcal{Q})$.

This simple rule is translated into pseudocode in Alg. 23.

Algorithm 23 The basic NaïvePartitionRefinement algorithm.

```
def NaïvePartitionRefinement($\mathcal{P}, f$):
  $\mathcal{Q} \leftarrow \mathcal{P}$
  while $\exists P \in \mathcal{Q} : \mathcal{Q} \neq \mathsf{split}\left(f^{-1}(P), \mathcal{Q}\right)$ :
    choose a splitter $P \in \mathcal{Q}$
    $\mathcal{Q} \leftarrow \mathsf{split}\left(f^{-1}(P), \mathcal{Q}\right)$          ▷The refinement step.
  return $\mathcal{Q}$
```

The refinement step in Alg. 23 replaces a partition which is *unstable* with respect to the preimage of some block P by a refinement with respect to the preimage of P. Due to the inheritance under refinement, any given set can only be used as a splitter once—after that, splitting with regards to it will have no effect.

We proceed to show the correctness of this simple algorithm. First we prove that the partition $\mathcal{Q}$ never gets *too* fine—it always stays at least as coarse as the coarsest stable refinement.

Lemma 3.11.8. *At any iteration of Alg. 23, the coarsest stable refinement of $\mathcal{R}$ of the original partition $\mathcal{P}$ is also a refinement of the current partition $\mathcal{Q}$, i.e., $\mathcal{R} \preccurlyeq \mathcal{Q}$.*

Proof. Let $\mathcal{R}$ be a coarsest stable refinement of $\mathcal{P}$. We prove that by induction on the number of iterations.

- **Base case**: Holds by definition.
- **Inductive step**: Suppose the lemma holds before a refinement step which splits the partition $\mathcal{Q}$ with a splitter S. Since S is a block in $\mathcal{Q}$ and, by our inductive hypothesis, $\mathcal{R}$ is a refinement of $\mathcal{Q}$, S is also a union of blocks in $\mathcal{R}$. By Lemma 3.11.5, a stable partition is stable with respect to the union of any subset of its blocks. Therefore, $\mathcal{R}$ is stable with regards to S. Since split is monotone by Lemma 3.11.6, $\mathcal{R} = \mathsf{split}(f^{-1}(S), \mathcal{R})$ is a refinement of $\mathsf{split}(f^{-1}(S), \mathcal{Q})$.

■

Besides being useful for the correctness of the simple partition refinement algorithm, Lemma 3.11.8 also enables us to show that coarsest stable refinements are in fact *unique*.

Theorem 3.11.4 (Berkholz et al. (2017))**.** *Let $\mathcal{P}$ be an initial partition of some set U and f a total function on U. Then, the coarsest stable refinement of $\mathcal{P}$ is* unique.

Proof. Assume the contrary: suppose there exists some partition $\mathcal{P}$ with two distinct coarsest stable refinements $\mathcal{Q}_1$ and $\mathcal{Q}_2$. Choose a partition $\mathcal{P}$ such that $|\mathcal{P}|$ is maximal. Then, $\mathcal{P}$ is not stable (otherwise, the *unique* coarsest stable refinement would be $\mathcal{P}$ itself). So there exists some splitter S which effectively splits $\mathcal{P}$ (it increases the number of block in the partition), i.e., $|\mathsf{split}\left(f^{-1}(S), \mathcal{P}\right)| > |\mathcal{P}|$. However, by Lemma 3.11.8, both $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are still refinements of $\mathsf{split}\left(f^{-1}(S), \mathcal{P}\right)$. However, this contradicts our assumption that $\mathcal{P}$ is the partition with the most blocks among all partitions of U with multiple coarsest stable refinements. ■

We now show that Alg. 23 terminates and due to the terminating condition returns the coarsest stable refinement. This will also give us a runtime bound.

Theorem 3.11.5 (Paige and Tarjan (1987))**.** *Alg. 23 terminates in at most $N - 1$ iterations and returns the coarsest stable refinement of $\mathcal{P}$ with respect to f. Moreover, that refinement is unique.*

Proof. Any partition $\mathcal{Q}$ of the set U can have *at most* N blocks. By construction, the number of blocks of $\mathcal{Q}$ in Alg. 23 *increases* after each iteration. Therefore, since $|\mathcal{P}| \geq 1$, the algorithm terminates in at most $N - 1$ iterations. Upon termination, Alg. 23 returns a stable refinement by the termination condition. Now, by Lemma 3.11.8, *any* coarsest stable refinement must itself be a refinement of the output. Thus, taking into account also Thm. 3.11.4 the algorithm returns the unique coarsest stable refinement. ■

Complexity. The naive partitioning algorithm can be implemented to run in time $\mathcal{O}\left(N^2\right)$ if we store, for each $u \in U$, its preimage $f^{-1}(\{u\})$. Importantly, splitting itself takes $\mathcal{O}\left(N\right)$. The final bound is the result of the bound on the number of iterations from Thm. 3.11.5.

Hopcroft's Algorithm for Partition Refinement

We now seek to improve the naïve above approach from above. The key step in speeding up the naïve construction is to make a simple observation: Once we have partitioned with a certain block S, we clearly do not have to split with it again, unless S gets split itself. However, in this case, we actually have an *easier* job: we only have to split with *one* of the subsets S was split into—the result of splitting on either is the same! The next lemma formalizes this.

Lemma 3.11.9 (Hopcroft (1971))**.** *Let $\mathcal{P}$ be a partition of U. Let S be a splitter and $\mathcal{Q} = \mathsf{split}(f^{-1}(S), \mathcal{P})$. Then for any $Q \in \mathcal{Q}$:*

$$\mathsf{split}(f^{-1}(S \setminus Q), \mathsf{split}(f^{-1}(Q), \mathcal{Q})) = \mathsf{split}(f^{-1}(Q), \mathcal{Q}). \tag{3.219}$$

Proof. Consider $B \in \mathsf{split}(f^{-1}(Q), \mathcal{Q})$.

- **Case 1:** $B \subseteq f^{-1}(Q)$, then $B \cap f^{-1}(S \setminus Q) = \emptyset$. So $\mathcal{Q}$ is stable with respect to $S \setminus Q$.
- **Case 2:** $B \subseteq f^{-1}(S) \setminus f^{-1}(Q)$, then $B \subseteq f^{-1}(S \setminus Q)$. So, again, $\mathcal{Q}$ is stable with respect to $S \setminus Q$

■

The lemma above shows that we only ever have to split with one of the created subblocks after we have already split with the entire block—we get the result of partitioning with the other subset for free. Given the free choice, for efficiency, we always choose to split with the *smaller* of the resulting splits.

This suggests the algorithm presented in Alg. 24. In contrast to the naïve approach above, Hopcroft now maintains a queue of possible future splitters, stack, and uses a modified function for splitting, termed split-modified, which is itself is presented in Alg. 25. split-modified splits all the blocks in the partition and modifies the queue of possible future splitters. It only stores the smaller of the resulting future splitters at each iteration, whenever the full block has been used for splitting before (Lines 11–16). This is marked by the fact it is not on stack anymore. If the block has not been used for splitting before, both subblocks are added to stack (Lines 8–10). The change to the splitting portion of the algorithm is the only on needed to get a significantly faster algorithm, as we show below.

Algorithm 24 The Hopcroft algorithm.

```
def Hopcroft($\mathcal{P}, f$):
   stack ← $[P \mid P \in \mathcal{P}]$                ▷Initialize a stack of possible splitters.
   while |stack| > 0 :
      pop $S$ from stack
      $\mathcal{P}$ ← split-modified $\left(f^{-1}(S), \mathcal{P}, \text{stack}\right)$
   return $\mathcal{P}$
```

Algorithm 25 The split-modified algorithm.

```
1.  def split-modified($f^{-1}(S), \mathcal{P}$, stack):
2.     $\mathcal{Q} \leftarrow \{\}$                                        ▷Initialize a new partition.
3.     for $P \in \mathcal{P}$ :
4.        ▷ Splitting the partition $\mathcal{P}$ with $f^{-1}(S)$
5.        if $|f^{-1}(S) \cap P| > 0$ and $|f^{-1}(S) \setminus P| > 0$ :
6.           push $f^{-1}(S) \cap P$ and $f^{-1}(S) \setminus P$ to $\mathcal{Q}$
7.           ▷ $f^{-1}(S) \cap P$ and $f^{-1}(S) \setminus P$ are new blocks in the refinement.
8.           if $P \in$ stack :
9.              ▷ $\mathcal{P}$ has not been split by $P$ yet.
10.             replace $P$ in stack with $f^{-1}(S) \cap P$ and $f^{-1}(S) \setminus P$
11.          else
12.             ▷ $\mathcal{P}$ has already been split by $P$. Only one of the subsets must be considered for splitting later.
13.             if $|f^{-1}(S) \cap P| < |f^{-1}(S) \setminus P|$ :
14.                push $f^{-1}(S) \cap P$ to stack
15.             else
16.                push $f^{-1}(S) \setminus P$ to stack
17.       else
18.          ▷ The unsplit $P$ remains in the refinement.
19.          push $P$ to $\mathcal{Q}$
20.    return $\mathcal{Q}$
```

Complexity. Intuitively, since we always choose the *smaller* candidate splitter at each iteration, the combined size of the entire set of splitters should decay exponentially. To analyze the complexity of Hopcroft in more detail, we make use of the following lemma.

Lemma 3.11.10. *Alg. 25 can be implemented in $\mathcal{O}\left(|f^{-1}(S)|\right)$ time.*

Proof. We have to consider a few implementation details not made explicit in the pseudocode to show that splitting can be done in linear time. To follow the proof more closely, the modified pseudocode is presented in Alg. 26. Firstly, we keep an auxiliary map, call it in-block, where in-block $[u]$ stores the block which $u \in U$ belongs to. in-block can trivially be populated in time $\mathcal{O}(N)$ (this is only executed once outside of the loop and does thus not affect the runtime of Alg. 24) at the beginning of the execution and updated on Line 6 in Alg. 25 in time proportional to the construction of the sets themselves. For any $P \in \mathcal{P}$, we can construct the sets $f^{-1}(S) \cap P$ and $f^{-1}(S) \setminus P$ in time $\mathcal{O}\left(|f^{-1}(S)|\right)$ by looping over $f^{-1}(S)$ *once* and splitting each $P \in \mathcal{P}$. Therefore, the aggregate runtime of the entire function call is $\mathcal{O}\left(|f^{-1}(S)|\right)$. ■

Algorithm 26 The FastHopcroft algorithm.

```
1.  def FastHopcroft($\mathcal{P}, f$):
2.    stack ← $[S_i \mid S_i \in \mathcal{P}]$
3.    in-block ← $\{u : S_i \ \forall u \in S_i, \forall S_i \in$ stack$\}$
4.    ▷ Empties in $\mathcal{O}(\log N)$ iterations.
5.    while |stack| $> 0$ :
6.      pop $S_i$ from stack
7.      inverse ← $f^{-1}(S_i)$                                        ▷$\mathcal{O}\left(|f^{-1}(S_i)|\right)$ time.
8.      $\ell$ ← [(in-block $[u]$, $u$) $\forall u \in$ inverse]
9.      count ← a map storing, $\forall U_j \in \mathcal{P}$, the number of its elements appearing in inverse
10.     covered ← $\{U_j \ \forall \ U_j$ such that count $[U_j] = |U_j|\}$
11.     $\mathcal{R} \leftarrow \{\}$
12.     for $U_j, u \in \ell$ such that $U_j \notin$ covered :
13.       remove $u$ from $U_j$
14.       add $u$ to
15.     for $R_j \in \mathcal{R}$ :
16.       for $u \in R_j$ :
17.         in-block $[u] \leftarrow R_j$
18.       add $R_j$ to stack
19.       add $R_j$ to $\mathcal{P}$
20.   return $\mathcal{P}$
```

Theorem 3.11.6. *Hopcroft runs in time $\mathcal{O}(N \log N)$.*

Proof. Consider an element $u \in U$. We first examine how many times a block containing u can be created and put on stack (we will say that u was put on stack) when u was previously *not* in any block on stack. This happens once at initialization on Line 2 in Alg. 24. It does *not* happen on Line 10 of Alg. 25 since P, which contains u, is *replaced*, meaning that u was *already* in a block on stack before. The only other case when we can add u to stack are Lines 14 and 16. If u is added to stack there, u is then in a block at most half the size of the block it was previously in. Therefore, we can conclude that u can be put on stack without it being there before *at most* $1 + \log N$ times.

Any time a block containing u is chosen as a splitter, it contributes a cost of $|f^{-1}(\{u\})|$ to the execution of the loop, since $f^{-1}(S) = \bigcup_{u \in S} f^{-1}(\{u\})$ where $f^{-1}(\{u\})$ is the pre-image of u under f. But as we argued above, u can not be in a splitter S popped from the stack more than $1 + \log N$. The total contribution of a $u \in U$ to the runtime of the algorithm is therefore $|f^{-1}(\{u\})| \cdot \mathcal{O}(\log N)$. Since $\sum_{u \in U} |f^{-1}(\{u\})| = N$, we can conclude that the total runtime of Alg. 24 is $\mathcal{O}(N \log N)$. ■

This concludes our treatment of Hopcroft for general partition refinement. We will return to it later to see how we can use it for automata minimization.

Moore's Algorithm as Partition Refinement

We introduced Moore's algorithm in the context of minimizing unweighted automata in Alg. 22. However, we can also write out Moore's algorithm more generally in the spirit of partition refinement. This version is presented in Alg. 27 and contains details of table-filling left out in the pseudocode of Moore. Here, we abstract away the transition function of the automaton to some general function f. However, the idea of the algorithm remains the same—it maintains a list of pairs we know to be distinguished by the function f, distinguished. This list is initialized with pairs in different blocks of the original partition in Lines 3–8. All pairs of elements q, q' are then processed to determine whether the function f maps them to distinguished pairs. If so, then the pair q, q' is distinguished as well. Not only that—all the pairs of elements which are (transitively) mapped to q, q' are distinguished too. Thus, the newly discovered change therefore has to be propagated back to all the states which eventually end up in q, q'. This is achieved by keeping an additional structure trace-back, which keeps, for and pair p, p' where $p \neq p'$, all the discovered pairs which eventually end up in p, p'. Upon discovering that q, q' are distinguished, we can then loop through their entry in trace-back and mark all pairs in there as distinguished as well. This is done on Lines 14–21. Once all pairs have been processed, we know all the pairs which are distinguished by f. As reasoned in the proof of Moore, all pairs which are not distinguished contain elements which belong to the same partition. Lines 26–37 therefore group equivalent elements into partitions together and construct the final partition which is stable with regards to f.

3.11.4 FSA Minimization as Partition Refinement

Now that we are familiar with partition refinement in general, we can discuss how FSA minimization can be seen as a specific application of it. We will see that we can present minimization as a simple loop calling partition refinement a number of times! Intuitively, we can think of the transition function δ defining the right languages of the states as the partitioning function f in partition refinement. However, δ is a function of two arguments—the current state and the *input symbol*—and it behaves differently depending on the input symbol. We therefore cannot use it directly as the function f in partition refinement. The solution, however, is trivial. As we will see, all we will have to do is run partition refinement *multiple times*, once for each possible input symbol. This will of course incur a constant factor in the runtime of the algorithm. Let us now explore the entire approach in more detail.

Construction of the Minimal FSA. The general idea of using partition refinement for automata minimization is to partition the state space into blocks of identical states—we will therefore operate directly on sets of states in the minimization algorithms themselves. The goal of minimization, however, is to produce a minimial *automaton*, not a partition of the states. Therefore, before describing the actual minimization is detail, we introduce a helper function for constructing a new, equivalent, FSA based on the equivalence classes constructed by minimization. The construction is

Algorithm 27 The MoorePartition algorithm.

```
def MoorePartition($\mathcal{P}, f$):
  all-pairs $\leftarrow \{(q, q') \mid q, q' \in Q\}$
  distinguished $\leftarrow \{\}$
  ▷ Initialize distinguished pairs (those belonging to different initial blocks).
  for $Q, Q' \in \mathcal{P}$ :
    if $Q \neq Q'$ :
      for $q \in Q, q' \in Q'$ :
        add $(q, q')$ to distinguished
  trace-back $\leftarrow \{\}$
  for $q, q' \in$ all-pairs :
    if $(f(q), f(q')) \in$ distinguished :                ▷$q$ and $q'$ are distinguished.
      add $(q, q')$ to distinguished
      ▷ Propagate the change.
      stack $\leftarrow \{(q, q')\}$
      while $|$stack$| > 0$ :
        pop $(p, p')$ off stack
        add $(p, p')$ to distinguished
        ▷ Unroll all the way back over the states which lead to the newly discovered distinguished pair.
        for $r, r' \in$ trace-back $[p, p']$ :
          if $(r, r') \notin$ distinguished :
            add $(r, r')$ to stack
    else
      if $f(q) \neq f(q')$ :
        ▷ Allows back-propagation of distinguished pairs if $(f(q), f(q'))$ are found to be distinguished.
        add $(q, q')$ to trace-back $[(f(q), f(q'))]$
  $\mu \leftarrow \{\}$                                   ▷Initialize the map of equivalence classes of the states.
  for $q \in Q$ :
    $\mu[q] \leftarrow \{q\}$
  for $q, q' \in$ all-pairs $\setminus$ distinguished :
    $N \leftarrow \mu[q] \cup \mu[q'] \cup \{q, q'\}$       ▷Combine equivalence classes of equivalent states.
    for $q'' \in N$ :                                    ▷Update the equivalence classes of all newly equivalent states.
      $\mu[q''] \leftarrow N$
  $\mathcal{Q} \leftarrow \{\}$                              ▷Initialize the final partition.
  ▷ Add the finalized equivalence classes to $\mathcal{Q}$.
  for $q \in Q$ :
    if $\mu[q] \notin \mathcal{Q}$ :
      add $\mu[q]$ to $\mathcal{Q}$
  return $\mathcal{Q}$
```

presented in Alg. 28. It receives a partition $\mathcal{P}$ of the state space Q and constructs a new FSA $\mathcal{A}_{\text{MIN}}$ whose states represent the *blocks* in the partition. The transitions between those states represent all possible transitions between equivalence classes, i.e., two classes are connected if any of the states within them are connected (Lines 7–9).[24] Finally, the initial and final states are added to $\mathcal{A}_{\text{MIN}}$. The former are the identified with the equivalence classes containing at least one initial state (Line 11) and the latter with the equivalence classes containing at least one final state (Line 13). By construction, all states within an equivalence class containing at least one final state are of course also final. Note that, for generality, the function is presented for the weighted case. We will reuse the same function for the weighted construction later. As always, you can just think of using the Boolean semiring when talking about the unweighted version of the function. We will use this helper function in the partition refinement-based algorithm for minimization we present next, after partition refinement is used to construct the family of equivalence states.

Algorithm 28 The BlockFSAConstruction algorithm.

```
1.  def BlockFSAConstruction($\mathcal{A}, \mathcal{P}$):
2.      $\mathcal{A}_{\text{MIN}} \leftarrow (\Sigma, Q_{\text{MIN}}, I_{\text{MIN}}, F_{\text{MIN}}, \delta_{\text{MIN}})$    ▷Initialize a new automaton over the same semiring as $\mathcal{A}$.
3.      $\mu \leftarrow \{\}$    ▷Initialize an empty map for bookkeeping.
4.      for $\mathcal{Q} \in \mathcal{P}$ :
5.          for $q \in \mathcal{Q}$ :
6.              $\mu[q] \leftarrow \mathcal{Q}$    ▷Link each state $\boldsymbol{q}$ to its equivalence class.
7.      for $q \in Q$ :
8.          for $q \xrightarrow{a/w} q' \in \mathcal{E}(q)$ :
9.              add transition $\mu[q] \xrightarrow{a/w} \mu[q']$ to $\mathcal{A}_{\text{MIN}}$
10.     for $q_I \in I$ :
11.         $\lambda_{\text{MIN}}(\mu[q_I]) \leftarrow \lambda(\mu[q_I]) \oplus \lambda(q_I)$
12.     for $q_F \in F$ :
13.         $\rho_{\text{MIN}}(\mu[q_F]) \leftarrow \rho(\mu[q_F]) \oplus \rho(q_F)$
14.     return $\mathcal{A}_{\text{MIN}}$
```

We now turn to the main point of this section—how to describe automata minimization as partition refinement. In particular, we will be able to use the faster Hopcroft's algorithm and arrive at an $\mathcal{O}(|\Sigma||Q|\log|Q|)$ algorithm for automaton minimization. We introduce the approach in the unweighted case, but, as we will see below, the generalization to the weighted case will not require us to change the actual algorithm at all. Thus, the completely general schema of how minimization can be done in the framework of partition refinement is presented in Alg. 29. It allows us to plug in *any* partition refinement procedure on Line 5, which is where we partition the set of states according to a function we prepare based on δ.

The general idea is the procedure is as follows. For automaton $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, we iteratively partition the states with regards to the transition function restricted to a certain $a \in \Sigma$, where restriction to a symbol $a \in \Sigma$ means that we construct a function f_a for each $a \in \Sigma$ mapping a state q to the target of its (unique) a-labeled transition. We start with the partition separating the final and non-final states on Line 2, which are, in an ε-free deterministic automaton, clearly distinguished. We then sequentially partition with regards to each f_a, taking the resulting partition of the previous

[24] Since we assume the partition $\mathcal{P}$ is the *coarsest* stable partition with regards to δ, such a definition is well-specified, i.e., if any state $q_i \in P_i$ has a a-transition into some state $q_j \in P_j$ for some $a \in \Sigma$, any other state $q'_i \in P_i$ will have such a transition into some state $q'_j \in P_j$ as well.

partitioning as the input for the partitioning with regards to the next function on Lines 5 and 6.

Algorithm 29 The PartitionRefinementMinimization algorithm. partition can be any partition refinement algorithm.

```
1. def PartitionRefinementMinimization($\mathcal{A}$):
2.     $\mathcal{P} \leftarrow \{F, Q \setminus F\}$                                    ▷Initialize the partition.
3.     for $a \in \Sigma$ :
4.         ▷ Initialize the function $f_a$ as the function mapping a state $q$ to the target of the a-transition from $q$.
5.         $f_a \leftarrow \left\{ q \mapsto q' \mid q \xrightarrow{a/\bullet} q' \in \mathcal{E}(q), q \in Q \right\}$
6.         $\mathcal{P} \leftarrow$ partition $(f_a, \mathcal{P})$
7.     return BlockFSAConstruction $(\mathcal{A}, \mathcal{P})$
```

The next theorem establishes that a partition constructed in this way is stable with regards to δ and is the coarsest possible among all such partitions.

Theorem 3.11.7. *Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be a FSA. Let $\mathcal{P}$ be an initial partition into* final *and* non-final *states and let $\mathcal{Q}$ be the resulting partition after executing the loop between Lines* 3 *and* 6 *in Alg. 29 for all $a \in \Sigma$. Then $\mathcal{Q}$ is the coarsest partition refinement of $\mathcal{P}$ with respect to δ. It partitions Q into blocks of equivalent states.*

Proof. Due to the monotonicity of splitting (Lemma 3.11.6), this procedure clearly produces a partition which is stable with regards to the entire function δ—no matter which sequence of symbols we consume, the partitioning with regards to all of them guarantees that the partition will be stable with regards to all. On the other hand, it is also easy to see that all the splits done are necessary. Since the partitioning done with regards to any of the input symbols $a \in \Sigma$ refines the input partition such that it is the *coarsest* stable refinement with regards to f_a, we see that any alternative partition would lead to distinguished states being in the same block. ■

And, finally, we can conclude that partition refinement based minimization groups together equivalent states.

Theorem 3.11.8. *As above, let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be a FSA. Let $\mathcal{P}$ be an initial partition into* final *and* non-final *states and let $\mathcal{Q}$ be the resulting partition after executing the loop between Lines* 3 *and* 6 *in Alg. 29 for all $a \in \Sigma$. Then $\mathcal{Q}$ partitions Q into blocks of equivalent states.*

Proof. First, suppose that states q, q' are *not* distinguished by Alg. 29, i.e., they are members of the same block in $\mathcal{Q}$. We have to show that they are then equivalent. We will show that by proving that any path starting any of the two states ends up in the same block in $\mathcal{Q}$ by inducting on the path length. **Base case**: $|\boldsymbol{\pi}| = 0$. Since q and q' are part of the same block by assumption, the base case holds. **Inductive step** Suppose the claim holds for any path of length less than n. Any path of length n can be decomposed into a subpath of length $n-1$ and its *final* transition. By our inductive hypothesis, we know that the subpath of length $n-1$ does not distinguish q and q', meaning that both states reached with any the first $n-1$ transitions of any path of length n will be in the same block. Now, since the final partition $\mathcal{Q}$ is stable with regards to f_a for any $a \in \Sigma$, *any* transition from two states in the same block in $\mathcal{Q}$ will lead to the same state by definition of stability. This means that any path of length n will lead to some states p, p' in the *same* block in $\mathcal{Q}$. Since $\mathcal{Q} \preccurlyeq \mathcal{P} = \{F, Q \setminus F\}$, p and p' are either both final or both non-final. This implies that no path in the automaton will distinguish the two states q, q'.

Next, suppose that states q, q' are marked as distinguished by Alg. 29. We have to show that they are indeed distinguishable, i.e., that there exists a string $\boldsymbol{y}$ such that, when following the mappings of $\boldsymbol{y}$-labeled functions from q and q', we end up in a partition containing of final states from exactly one of the two states. Suppose the contrary, i.e., for any sequence of symbols $a_1, \ldots, a_n \in \Sigma$, following mappings labeled with $a_1 \ldots a_n$ from q leads to a block containing final states if and only if it leads to a block containing final states if and only if from q'. Then, the blocks of q and q', together with the blocks reachable from their blocks in the same number of mappings, could be *merged* into a common block and the partition would still be stable with respect to all f_a. However, since we know that $\mathcal{Q}$ is the *coarsest* stable partition of Q with respect to δ, this brings us to a contradiction, meaning that q and q' are indeed distinguished. This concludes the proof. ■

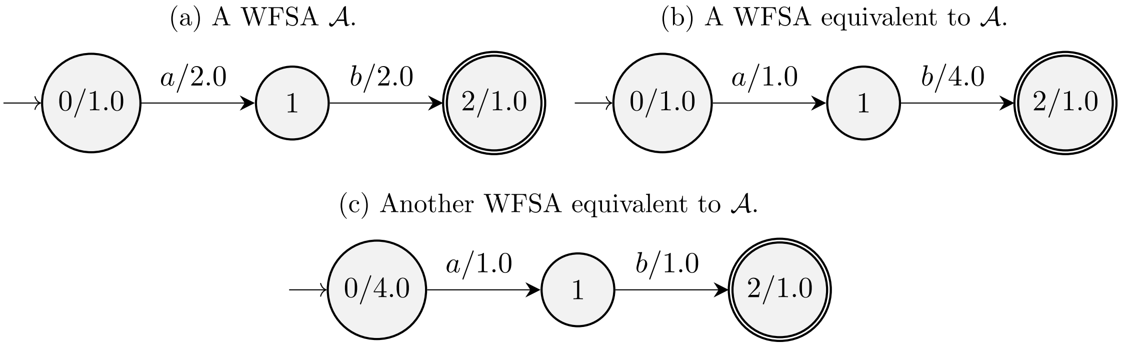
This gives us a very general way to minimize any unweighted FSA. Clearly, the runtime of minimization with partition refinement is $\mathcal{O}\left(|\Sigma| F(|Q|)\right)$, where $F(|Q|)$ is the complexity of partition refinement algorithm used. Particularly, if we use Hopcroft as the partition function, we get a total runtime of $\mathcal{O}\left(|\Sigma||Q| \log |Q|\right)$, improving the runtime of Moore.

This finishes our discussion of *unweighted* minimization. We now move to the case of weighed automata.

3.11.5 Weight Pushing

We now introduce the operation of weight pushing which serves to find a canonical distribution of the weights of a WFSA. It does this by manipulating the individual arc weight such that the pathsum is unaltered. We start our development of the weight pushing algorithm by observing that the distribution of the weights along accepting paths of an automaton does not affect the formal power series realized by that automaton. Indeed, this is a trivial consequence of the fact that a formal power series operates over strings. In practice, however, this means that even if we hold the topology of the WFSA fixed, there are an infinite number of WFSA that represent the same language. This makes it difficult to know whether two WFSA represent the same weighted language unless we have a mechanism to canonicalize the weights. And this is exactly the problem that weight pushing seeks to fix.

Example 3.11.5. *As an example, consider the simple linear-chain WFSA over the real semiring with one possible accepted string, "ab", shown in Fig. 3.34a. The weight of the string "ab" is* 4*, which we get by multiplying the initial and final weights on the path, together with the weights of the transitions along it. However, we could get the same weights by rearranging the weights in many different ways, for example, as shown in Fig. 3.34b and Fig. 3.34c. Notice that we can redistribute the weights either over the transitions* or *the initial and final weights.*



(a) A WFSA $\mathcal{A}$.

(b) A WFSA equivalent to $\mathcal{A}$.

(c) Another WFSA equivalent to $\mathcal{A}$.

Figure 3.34: Three linear chain WFSA with different distributions of the weights over the transitions and initial/final states which all express exactly the same weighted regular language.

Although the distribution of the weights may not affect the language accepted by an automaton, it can affect the efficiency of some pruning-based approaches, for example in speech recognition (Mohri and Riley, 2001).

Similarly to section §3.10, we will be working with *semifields* (see Def. 3.10.2).[25] Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a trim WFSA over a semifield $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$. Weight pushing redistributes the weights in $\mathcal{A}$ according to some $\mathbb{K}$-valued function $\Psi : Q \to \mathbb{K}$, which we will call a **potential function**. We don't make any assumptions about Ψ, except that we require that $\Psi(q) \neq \mathbf{0}\, \forall q \in Q$. Weight pushing then performs the following transformation of the weights:

$$\forall q_I \in I, \qquad \lambda_{\text{PUSH}}\left(q_I\right) \leftarrow \lambda\left(q_I\right) \otimes \Psi\left(q_I\right) \tag{3.220}$$

$$\forall e = q \xrightarrow{a/w} q' \in \delta, \qquad w_{\text{PUSH}}\left(e\right) \leftarrow \Psi\left(q\right)^{-1} \otimes w \otimes \Psi\left(q'\right) \tag{3.221}$$

$$\forall q_F \in F, \qquad \rho_{\text{PUSH}}\left(q_F\right) \leftarrow \Psi\left(q_F\right)^{-1} \otimes \rho\left(q_F\right) \tag{3.222}$$

[25] The idea of weight pushing can, however, also be generalized to weakly divisible semirings, similarly to weighted determinization.

The way that the new weights are defined means that, along any individual path, the values of the potential function cancel out in the multiplication and we are left with the original weight of the path. This means that the weight pushing results in an equivalent WFSA, as the following theorem formalizes.[26]

Theorem 3.11.9. *Let $\mathcal{A}_{\text{PUSH}} = (\Sigma, Q, \delta_{\text{PUSH}}, \lambda_{\text{PUSH}}, \rho_{\text{PUSH}})$ be the result of* WeightPush *applied to the automaton $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$. The weight of any successful path $\boldsymbol{\pi}$ from $\mathcal{A}$ is unchanged in $\mathcal{A}_{\text{PUSH}}$:*

$$\lambda_{\text{PUSH}}\left(p\left(\boldsymbol{\pi}\right)\right) \otimes w_{\text{PUSH}}\left(\boldsymbol{\pi}\right) \otimes \rho_{\text{PUSH}}\left(n\left(\boldsymbol{\pi}\right)\right) = \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes w\left(\boldsymbol{\pi}\right) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) \tag{3.223}$$

Proof. First, we note that $\mathcal{A}_{\text{PUSH}}$ has the same topology as $\mathcal{A}$. Let

$$\boldsymbol{\pi} = q_1 \xrightarrow{a_1/w_1} q_2, q_2 \xrightarrow{a_2/w_2} q_3, \dots, q_{n-1} \xrightarrow{a_{n-1}/w_{n-1}} q_n = e_1, e_2, \dots, e_{n-1} \tag{3.224}$$

be an arbitrary accepting path. We will show that its weight, multiplied with the initial and starting weights, remains unchanged. This is easy to see since for any $i = 1, \dots, n-2$, we have:

$$\begin{aligned} w_{\text{PUSH}}\left(e_i\right) \otimes w_{\text{PUSH}}\left(e_{i+1}\right) &= \Psi\left(q_i\right)^{-1} \otimes w\left(e_i\right) \otimes \Psi\left(q_{i+1}\right) \otimes \Psi\left(q_{i+1}\right)^{-1} \otimes w\left(e_{i+1}\right) \otimes \Psi\left(q_{i+2}\right) && (3.225) \\ &= \Psi\left(q_i\right)^{-1} \otimes w\left(e_i\right) \otimes w\left(e_{i+1}\right) \otimes \Psi\left(q_{i+2}\right). && (3.226) \end{aligned}$$

where we used the cancellative property of the inverse for the second equality. Furthermore, we have

$$\begin{aligned} \lambda_{\text{PUSH}}\left(q_1\right) \otimes w_{\text{PUSH}}\left(e_1\right) &= \lambda\left(q_1\right) \otimes \Psi\left(q_1\right) \otimes \Psi\left(q_1\right)^{-1} \otimes w\left(e_1\right) \otimes \Psi\left(q_2\right) && (3.227) \\ &= \lambda\left(q_1\right) \otimes w\left(e_1\right) \otimes \Psi\left(q_2\right) \qquad \text{(inverse)} && (3.228) \end{aligned}$$

and

$$\begin{aligned} w_{\text{PUSH}}\left(e_{n-1}\right) \otimes \rho_{\text{PUSH}}\left(q_n\right) &= \Psi\left(q_{n-1}\right)^{-1} \otimes w\left(e_{n-1}\right) \otimes \Psi\left(q_n\right) \otimes \Psi\left(q_n\right)^{-1} \otimes \rho\left(q_n\right) && (3.229) \\ &= \Psi\left(q_{n-1}\right)^{-1} \otimes w\left(e_{n-1}\right) \otimes \rho\left(q_n\right) \qquad \text{(inverse)} && (3.230) \end{aligned}$$

Combining the three equalities above, it is easy to show that the weight of the path $\boldsymbol{\pi}$ is preserved by weight pushing as the effect of the potential function is canceled out. Exercise Exercise 3.17 asks you to complete the proof. ■

The theorem above shows the correctness of the procedure with any non-zero valued potential function Ψ. A very natural question to ask now is how to choose Ψ such that the distribution of weights in the pushed automaton has some useful properties. One common choice is the backward weight function which we first saw in §3.6.2. Recall the definition of $\boldsymbol{\beta}$ of states $q \in Q$: the backward weight $\boldsymbol{\beta}\left(q\right)$ is the sum of all the path weights from q to a final state:

$$\boldsymbol{\beta}(q) = \rho(q) \oplus \bigoplus_{m=1}^{M} w_m \otimes \boldsymbol{\beta}(q_m) \tag{3.231}$$

We initially computed the values $\boldsymbol{\beta}$ only in *acyclic* graphs, but we can also compute them in the cyclic case with the only difference being that the values now have to be computed with Lehmann

[26] The result is, in fact, even somewhat stronger—not only are the string weights unaltered (which is required for equivalence), but the weights of the paths themselves don't change either. This means that weight pushing is a so-called strongly semantics preserving transformation. We will see more on that later in **??**.

instead of backward. By using the backward weights as the potential function, the algorithm effectively pushes the weights of each path as far as possible toward the initial states.

Intuitively, we can understand the weight pushing with the backward weights $\boldsymbol{\beta}$ as follows. The initial states are reweighted by $\otimes$-multiplying their initial weights with their $\boldsymbol{\beta}$-values, which means that, for any accepting path starting in a particular initial state q_I, it only has to account for the "fraction" of the total weight $\boldsymbol{\beta}(q_I)$ it contributes on its own. For each edge, we reweight it by first pre-$\otimes$-multiplying the original weight with the inverse of the $\boldsymbol{\beta}$-value of its source state to "remove" the weight already accounted for in the previous transitions and the initial weight. Then, we post-$\otimes$-multiply it with the $\boldsymbol{\beta}$ of its target state to "push back" towards the start state as much of the remaining weight (in terms of the $\oplus$-sum of all the emanating paths from the target) stemming from the target state as possible. In the end, we again pre-$\otimes$-multiply the final values of all final nodes with the inverse of their $\boldsymbol{\beta}$-value, again to account for all the weight that has already been pushed forward. There is another useful feature of the running weight pushing with $\Psi = \boldsymbol{\beta}$: it makes the automaton **stochastic**, as the following result shows.

Theorem 3.11.10. *Let $\mathcal{A}_{\text{PUSH}} = (\Sigma, Q, \delta_{\text{PUSH}}, \lambda_{\text{PUSH}}, \rho_{\text{PUSH}})$ be the result of WeightPush applied to the automaton $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$. If we choose the potential function Ψ to be the backward weight $\boldsymbol{\beta}$, the WFSA $\mathcal{A}_{\text{PUSH}}$ is **stochastic**, i.e.,*

$$\forall q \in Q, \quad \bigoplus_{q \xrightarrow{\bullet/w} q' \in \mathcal{E}_{\text{PUSH}}(q)} w \oplus \rho(q) = \boldsymbol{1}. \tag{3.232}$$

where $\mathcal{E}$ are all outgoing edges from q.

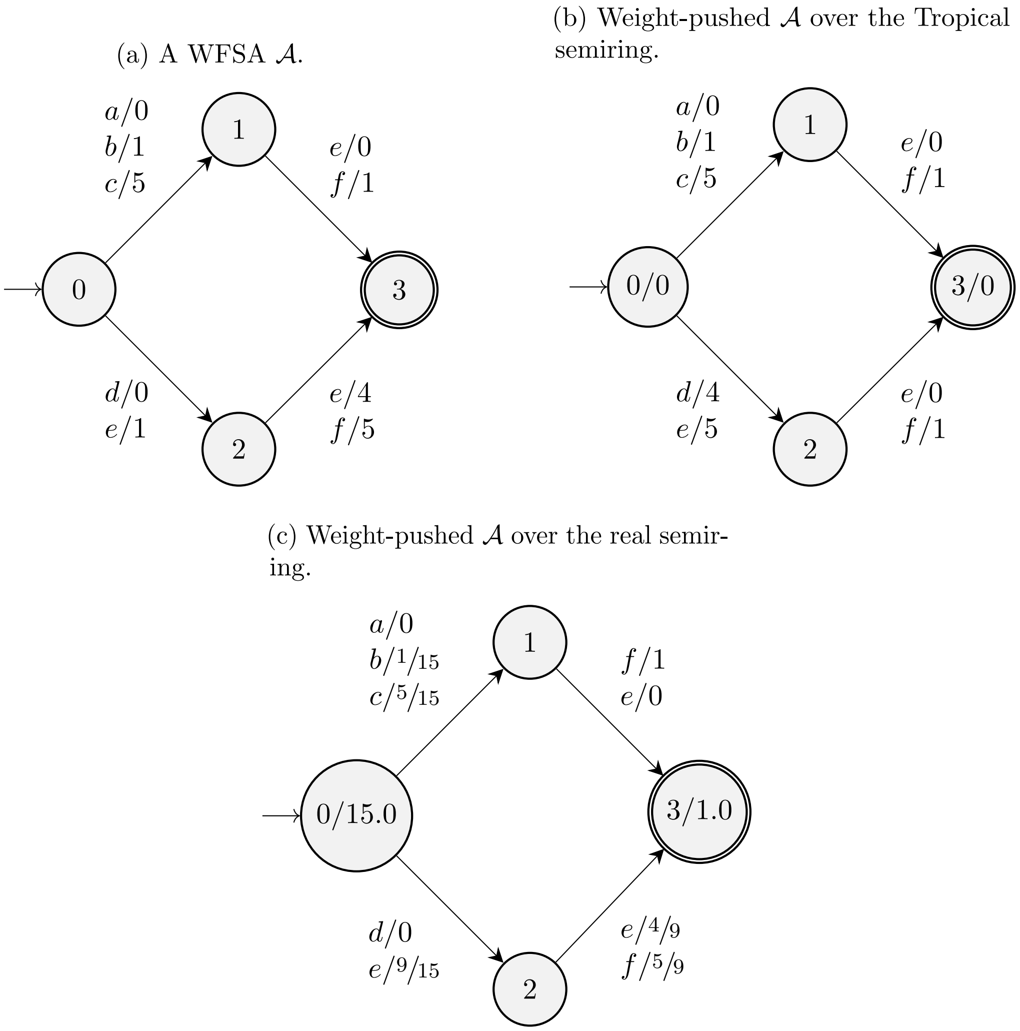
Proof. Let $q \in Q$ such that $\boldsymbol{\beta}(q) \neq \mathbf{0}$. Then

$$\begin{aligned}
\bigoplus_{q \xrightarrow{\bullet/w_{\text{PUSH}}} q' \in \mathcal{E}_{\text{PUSH}}(q)} w_{\text{PUSH}} \oplus \rho_{\text{PUSH}}(q) &= \bigoplus_{q \xrightarrow{\bullet/w} q' \in \mathcal{E}(q)} \boldsymbol{\beta}(q)^{-1} \otimes w \otimes \boldsymbol{\beta}(q') \oplus \boldsymbol{\beta}(q)^{-1} \otimes \rho(q) && \text{(definition)} \\
&= \boldsymbol{\beta}(q)^{-1} \otimes \bigoplus_{q \xrightarrow{\bullet/w} q' \in \mathcal{E}(q)} w \otimes \boldsymbol{\beta}(q') \oplus \rho(q) && \text{(distributivity)} \\
&= \boldsymbol{\beta}(q)^{-1} \otimes \boldsymbol{\beta}_(q) && \text{(definition)} \\
&= \mathbf{1} && \text{(inverse)}
\end{aligned}$$

■

This means that one can imagine the weights of the outgoing transitions of any state $q \in Q$ as forming a "probability distribution" over the transitions. The quotation marks come from the fact that the multiplicative inverse depends on the semiring in use. However, if the automaton $\mathcal{A}$ is defined over the semiring of the positive reals, weight pushing does just that—it defines a valid probability distribution over the outgoing transitions of any state. You can therefore think of weight pushing with $\Psi = \boldsymbol{\beta}$ as a very general way of *locally* normalizing a *globally* normalized model! The following example will hopefully help to firm up the intuition.

Example 3.11.6. *Let us demonstrate weight pushing on a simple example from Mohri (2009). Fig. 3.35 shows a WFSA over either the real or Tropical semiring and two equivalent WFSA with their weights pushed, one over the real and one over the Tropical semiring.*



(a) A WFSA $\mathcal{A}$.

(b) Weight-pushed $\mathcal{A}$ over the Tropical semiring.

(c) Weight-pushed $\mathcal{A}$ over the real semiring.

Figure 3.35: An example of a WFSA and two weight-pushed variants over different semirings from Mohri (2009). Notice the different role of the number 0 in both semirings.

To run weight pushing with the backward weights as the potentials, we therefore need to calculate the backward weights $\boldsymbol{\beta}$ for all states. We can do this in any of the ways we saw in §3.6.2! For example, we could compute the entire matrix of the pathsums between state pairs and then use that to obtain the backward weights. Then, we just run the above operations to reweight the transitions and the initial/final weights. The complete algorithm is presented in Alg. 30.

By canonicalizing the distribution of the weights, weight pushing can be used to asses the equivalence of *deterministic* WFSA given an algorithm to test unweighted DFA equivalence: Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two WFSA and $\mathcal{A}_1'$ and $\mathcal{A}_2'$ be their weight-pushed variants. Then, we can determine whether they are equivalent by running an equivalence algorithm for *unweighted* automata on $\mathcal{A}_1'$ and $\mathcal{A}_2'$, where we consider each pair (transition label, transition weight) as a single compound label. This only works for deterministic automata. If we start with automata which are non-deterministic but determinizable, we can determinize them and run weight pushing and equivalence testing on those. However, as we saw in section §3.10.9, not all WFSA are determinizable.

Algorithm 30 The WeightPush algorithm with $\Psi = \boldsymbol{\beta}$. backward-values can be implemented using any appropriate algorithm for computing backward values.

```
def WeightPush($\mathcal{A}$):
    $\boldsymbol{\beta} \leftarrow$ backward-values $(\mathcal{A})$    ▷General pathsum to compute backward values.
    $\mathcal{A}_{\text{PUSH}} \leftarrow (\Sigma, Q, \delta_{\text{PUSH}}, \lambda_{\text{PUSH}}, \rho_{\text{PUSH}})$    ▷Initialize the automaton $\mathcal{A}_{\text{PUSH}}$ over the same semiring as $\mathcal{A}$
    for $q \in Q$ :
        $\lambda_{\text{PUSH}}(q) \leftarrow \lambda(q) \otimes \boldsymbol{\beta}(q)$
        $\rho_{\text{PUSH}}(q) \leftarrow \boldsymbol{\beta}(q)^{-1} \otimes \rho(q)$
        for $q \xrightarrow{w/a} q' \in \mathcal{E}(q)$ :
            add arc $q \xrightarrow{a/\boldsymbol{\beta}(q)^{-1} \otimes w \otimes \boldsymbol{\beta}(q')} q'$ to $\mathcal{A}_{\text{PUSH}}$
    return $\mathcal{A}_{\text{PUSH}}$
```

Complexity The complexity of the weight pushing algorithm heavily depends on the complexity of calculating the values of the potential function. In the case when $\Psi = \boldsymbol{\beta}$, the algorithm has to compute the backward weights in a general WFSA. Therefore, the runtime will be at most cubic in the number of states, $\mathcal{O}\left(|Q|^3\right)$. After the backward values have been computed, we just loop over all edges of the automaton, which takes $\mathcal{O}(|\delta|)$. Therefore, the entire runtime is dominated by the computation of the backward weights, and so the algorithm runs in $\mathcal{O}\left(|Q|^3\right)$.

Applications of weight pushing Let us finish by saying that weight pushing for canonicalization of WFSA is by far not the only useful application of weight pushing. We focus on it here due to its usefulness for weighted minimization, which we discuss in the next section, and due to its intuitive interpretation developed in Thm. 3.11.10. We will see another use of weight pushing with the *forward* weights $\boldsymbol{\alpha}$ in the next chapter. Uses of weight pushing also span into seemingly unrelated fields, for example, model-based reinforcement learning. There, potential-based reward shaping (Ng, 2003) is a form of weight pushing as well!

3.11.6 Weighted FSA Minimization

This section generalizes the algorithms for minimization we developed so far to the weighted case. As before, we will be working with *deterministic* automata, which we now also assume are *trim*. As in the unweighted case, a weighted automaton is minimal if has the smallest number of states among all equivalent WFSA. As we will see, a minimal WFSA may not be *unique*, but the topology of all minimal WFSA is identical.

Luckily, minimization of weighted automata is not much different from the unweighted case. In fact, we will not have to come up with any new algorithms for it, but rather use the ones we have developed so far smartly.

We start by defining the weighted equivalence of right languages.

Definition 3.11.11 (Language Equivalence)**.** *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA over some divisible semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ and $q, q' \in Q$ two states. We say that the weighted right language of q, L, and the weighted right language of q', L', are* ***equivalent*** *if and only if $\exists k, k' \in \mathbb{K}$ such that*

$$k^{-1} \otimes L = k'^{-1} \otimes L', \tag{3.233}$$

where the symbol '=' refers to the identity of languages as defined in Def. 3.9.2 in §3.9. Here, we define the scalar multiplication of a language L as the scalar multiplication of the formal power series representing the language.

We then extend this definition to the notion of equivalent *states*.

Definition 3.11.12 (Equivalence of States)**.** *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA. Two states $q, q' \in Q$ are* ***equivalent*** *if their right languages are equivalent. We will denote the equivalence class of a state q as $[q]$.*

If two states are equivalent, the "constant factors" which differentiate their otherwise identical languages will be pushed forward towards the initial states after weight pushing with $\Psi = \boldsymbol{\beta}$, leaving the weights of the edges *identical*. Any equivalent states will therefore become identical, meaning that we can consider the transition weights leading from them to some final states jointly with the input symbols and minimize based on those together.

Reducing Weighted Minimization to the Unweighted Case

To perform weighted minimization, we will not have to develop any new algorithms at all. Rather, the approach we pursue is to pre-process a weighted automaton such that the application of any unweighted minimization algorithm will correctly minimize it. Intriguingly, we already have all the machinery needed for such pre-processing. Namely, we will show in this section that *weight pushing* with backward weights as introduced in the beginning of the chapter is enough to modify any suitable WFSA into a form which allows very simple weighted minimization. Once a WFSA has been pushed, all that remains to be done is to minimize it with any unweighted minimization algorithm while treating the pairs of input symbols and (pushed) weights as input symbols of a (big) alphabet.

The reason this makes sense is intuitive. As in the unweighted case, we group together equivalent states.[27] As we discussed in the previous chapter, weight pushing *only* changes the distribution of the weights along the paths of a WFSA. In that way, it canonicalizes the distribution of the weights

[27] Exactly what this equivalence corresponds to will be defined shortly, but roughly you can think of it as having the same right languages modulo some constant multiplication factor of the weights.

by "pushing" them as close to the initial states as possible, and removing the "constant factors" which might distinguish states.

We now start developing some theory about equivalent states of WFSA which will allow us to characterize the minimal WFSA. We start with the following lemma.

Theorem 3.11.11 (Weighted Myhill-Nerode)**.** *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a deterministic WFSA and $L = L(\mathcal{A})$ its language. Then R_L has a finite number of equivalence classes and this number is less than or equal to the number of states of any WFSA* equivalent *to $\mathcal{A}$.*

Proof. Note that $\mathcal{A}$ only has 1 initial state since it is deterministic. Call it q_I. Let $\boldsymbol{x}, \boldsymbol{y} \in P_L$ be two strings and $\boldsymbol{\pi_x}$ and $\boldsymbol{\pi_y}$ the paths in the automata with yields $\boldsymbol{x}$ and $\boldsymbol{y}$ respectively. Then, we can choose $k = \lambda(q_I) \otimes w(\boldsymbol{\pi_x})$ and $k' = \lambda(q_I) \otimes w(\boldsymbol{\pi_y})$ as the division factors of their languages. This choice implies that

$$n(\boldsymbol{\pi_x}) = n(\boldsymbol{\pi_y}) \implies \boldsymbol{x} R_L \boldsymbol{y}, \tag{3.234}$$

i.e., as soon as the two strings lead to the same state, they are equivalent. That is because the division factors k and k' ensure the languages of $n(\boldsymbol{\pi_x})$ and $n(\boldsymbol{\pi_y})$ are identical. On the other hand, the relation

$$\boldsymbol{x} R \boldsymbol{y} \iff n(\boldsymbol{\pi_x}) = n(\boldsymbol{\pi_y}) \tag{3.235}$$

also defines an separate equivalence relation on P_L. The equivalence classes of this relation R consist of *states* of $\mathcal{A}$. The implication in Eq. (3.234) means that the number of equivalence classes of R (number of states) must be at least the number of equivalence classes of R_L, which proves the lemma. ■

To see the role of weight pushing in weighted minimization, consider $\mathcal{A}_{\text{PUSH}} = \text{WeightPush}(\mathcal{A})$. From Thm. 3.11.9, we know that $\mathcal{A}_{\text{PUSH}}$ is equivalent to $\mathcal{A}$ and it only differs in terms of the distribution of the weights over the paths of the automaton.

We now look at the automaton $\mathcal{A}_{\text{MIN}}$, obtained by running a minimization algorithm such as Hopcroft on automaton $\widetilde{\mathcal{A}_{\text{PUSH}}}$, where $\widetilde{\mathcal{A}_{\text{PUSH}}}$ is an *unweighted* automaton with the input symbols corresponding to the pairs of original input symbols and pushed weights, i.e., pairs (a, w), $a \in \Sigma$, $w \in \mathbb{K}$. As shown in the previous section, minimization algorithm will merge identical states of $\widetilde{\mathcal{A}_{\text{PUSH}}}$. The rest of the section shows that $\mathcal{A}_{\text{MIN}}$ constructed this way is indeed a minimal WFSA equivalent to $\mathcal{A}$.

Lemma 3.11.11. *Let $\mathcal{A} = (\Sigma, Q, I_1, F_1, \delta_1, \lambda_1, \rho_1)$ be a WFSA and $\mathcal{A}_{\text{PUSH}} = (\Sigma, Q, I_2, F_2, \delta_2, \lambda_2, \rho_2)$ be its pushed variant. Then for any two states $q, q' \in Q$, $[q] = [q']$ (q being equivalent to q') implies the following:*

$$\forall \boldsymbol{y} \in \Sigma^* : n(\boldsymbol{\pi}(q, \boldsymbol{y})) \in F \implies \boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}(q, \boldsymbol{y}) = \boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}(q', \boldsymbol{y}). \tag{3.236}$$

Proof. Note that $\boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}(q, \boldsymbol{y})$ is the weight of exactly one path (the one starting in q and yielding $\boldsymbol{y}$) due to the determinism of $\mathcal{A}_{\text{PUSH}}$. Let $\boldsymbol{y} = s_1 \dots s_n$ be a string of length $|\boldsymbol{y}| = n$, $\boldsymbol{\pi}_{q,\boldsymbol{y}} = q_0 = q \xrightarrow{s_1/w_1} q_1, q_1 \xrightarrow{s_2/w_2} q_2, \dots, q_{n-1} \xrightarrow{s_n/w_n} q_F$ be the $\boldsymbol{y}$-yielding path from the state q to a final state $q_F \in F$. Similarly, let $\boldsymbol{\pi}_{q'=q',\boldsymbol{y}} = q'_0 = q' \xrightarrow{s'_1/w'_1} q'_1, q-1 \xrightarrow{s'_2/w'_2} q'_2, \dots, q'_{n-1} \xrightarrow{s'_n/w'_n} q_F{}'$ be the $\boldsymbol{y}$-yielding path from the state q' to a final state $q_F{}' \in F$. Since $\mathcal{A}$ is deterministic, we have

$$\boldsymbol{\beta}_{\mathcal{A}}(q, \boldsymbol{y}) = w_1 \otimes \dots \otimes w_n \tag{3.237}$$

and

$$\boldsymbol{\beta}_{\mathcal{A}}(q', \boldsymbol{y}) = w'_1 \otimes \dots \otimes w'_n. \tag{3.238}$$

We will denote the weights along the $\boldsymbol{y}$-yielding paths from q and q' in the *pushed* automaton with v_i and v'_i, respectively. We have to show that, for any string $\boldsymbol{y} \in \Sigma^*$,

$$\boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}(q, \boldsymbol{y}) = v_1 \otimes \ldots \otimes v_n = v'_1 \otimes \ldots \otimes v'_n = \boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}\left(q', \boldsymbol{y}\right). \tag{3.239}$$

By definition, $[q] = [q']$ means that there exist $k, k' \in \mathbb{K}$ such that

$$k^{-1} \otimes L_q = k'^{-1} \otimes L_{q'} \tag{3.240}$$

or, equivalently,

$$L_{q'} = k'^{-1} \otimes k \otimes L_q. \tag{3.241}$$

Specifically, this implies the following two identities:

$$\boldsymbol{\beta}_{\mathcal{A}}\left(q'\right) = k'^{-1} \otimes k \otimes \boldsymbol{\beta}_{\mathcal{A}}(q) \tag{3.242}$$

and, even more specifically, $\forall \boldsymbol{y} \in \Sigma^*$,

$$\boldsymbol{\beta}_{\mathcal{A}}\left(q', \boldsymbol{y}\right) = k'^{-1} \otimes k \otimes \boldsymbol{\beta}_{\mathcal{A}}(q, \boldsymbol{y}). \tag{3.243}$$

Weight pushing modifies the weights w_i and w'_i, $i = 1, \ldots, n$ as follows

$$v_i \leftarrow \boldsymbol{\beta}_{\mathcal{A}}(q_{i-1})^{-1} \otimes w_i \otimes \boldsymbol{\beta}_{\mathcal{A}}(q_i) \tag{3.244}$$

$$v'_i \leftarrow \boldsymbol{\beta}_{\mathcal{A}}\left(q'_{i-1}\right)^{-1} \otimes w'_i \otimes \boldsymbol{\beta}_{\mathcal{A}}\left(q'_i\right). \tag{3.245}$$

Therefore, we have

$$\begin{aligned}
\boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}(q, \boldsymbol{y}) &= v_1 \otimes \ldots \otimes v_n && (3.246)\\
&= \boldsymbol{\beta}_{\mathcal{A}}(q)^{-1} \otimes w_1 \otimes \boldsymbol{\beta}_{\mathcal{A}}(q_1) \otimes \boldsymbol{\beta}_{\mathcal{A}}(q_1)^{-1} \otimes w_2 \otimes \boldsymbol{\beta}_{\mathcal{A}}(q_2) \otimes && (3.247)\\
&\quad \cdots \otimes \boldsymbol{\beta}_{\mathcal{A}}(q_{n-1})^{-1} \otimes w_n \otimes \boldsymbol{\beta}_{\mathcal{A}}(q_n) && (3.248)\\
&= \boldsymbol{\beta}_{\mathcal{A}}(q)^{-1} \otimes w_1 \otimes w_2 \otimes \ldots \otimes w_n \otimes \boldsymbol{\beta}_{\mathcal{A}}(q_n) && (3.249)\\
&= \boldsymbol{\beta}_{\mathcal{A}}(q)^{-1} \otimes \boldsymbol{\beta}_{\mathcal{A}}(q, \boldsymbol{y}) && (3.250)
\end{aligned}$$

and, similarly,

$$\boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}\left(q', \boldsymbol{y}\right) = \boldsymbol{\beta}_{\mathcal{A}}\left(q'\right)^{-1} \otimes \boldsymbol{\beta}_{\mathcal{A}}\left(q', \boldsymbol{y}\right). \tag{3.251}$$

Inserting Eq. (3.242) and Eq. (3.243) into the above, we get

$$\begin{aligned}
\boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}\left(q', \boldsymbol{y}\right) &= \left(k'^{-1} \otimes k \otimes \boldsymbol{\beta}_{\mathcal{A}}(q)\right)^{-1} \otimes k'^{-1} \otimes k \otimes \boldsymbol{\beta}_{\mathcal{A}}(q, \boldsymbol{y}) && (3.252)\\
&= \boldsymbol{\beta}_{\mathcal{A}}(q)^{-1} \otimes k^{-1} \otimes k' \otimes k'^{-1} \otimes k \otimes \boldsymbol{\beta}_{\mathcal{A}}(q, \boldsymbol{y}) && (3.253)\\
&= \boldsymbol{\beta}_{\mathcal{A}}(q)^{-1} \otimes \boldsymbol{\beta}_{\mathcal{A}}(q, \boldsymbol{y}) && (3.254)\\
&= \boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}(q, \boldsymbol{y}) && (3.255)
\end{aligned}$$

which concludes the proof. ■

Theorem 3.11.12. *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)_1$ be a WFSA and $q, q' \in Q$. If q and q' are* equivalent, *their right language is* identical *after weight pushing.*

Proof. Let $\mathcal{A}_{\text{PUSH}} = (\Sigma, Q, \delta, \lambda, \rho)_2$ be the automaton obtained by weight-pushing on $\mathcal{A}$, $L = L(\mathcal{A})$ and let $\boldsymbol{x}, \boldsymbol{y} \in P_L$ be the strings which lead to the states q and q' respectively.

For any $i = 0, \ldots, n-1$, $\boldsymbol{x} R_L \boldsymbol{y}$ implies $\boldsymbol{x} a_0 a_1 \ldots a_i R_L \boldsymbol{y} a_0 a_1 \ldots a_i$ for any $a_0 a_1 \ldots a_i \in \Sigma^*$ whenever $n\left(\boldsymbol{\pi}\left(q, \boldsymbol{x}\right)\right) \in F$. Therefore, by Lemma 3.11.11 we have $\forall i = 0, \ldots n-1$

$$\boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}\left(n\left(\boldsymbol{\pi}_{q,a_0a_1\ldots a_i}\right), a_{i+1}\ldots a_n\right) = \boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}\left(n\left(\boldsymbol{\pi}_{q',a_0a_1\ldots a_i}\right), a_{i+1}\ldots a_n\right), \tag{3.256}$$

which is, because $\mathcal{A}_{\text{PUSH}}$ is pushed, equivalent to

$$\boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}\left(n\left(\boldsymbol{\pi}_{q,a_0a_1\ldots a_i}\right), a_{i+1}\right) = \boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}\left(n\left(\boldsymbol{\pi}_{q',a_0a_1\ldots a_i}\right), a_{i+1}\right) \qquad \forall i = 1, \ldots, n-1. \tag{3.257}$$

This implies their right languages are identical, finishing the proof. ■

This finally suggests the construction summarized in the following theorem.

Theorem 3.11.13. *Given a trim, deterministic WFSA $\mathcal{A}$, a minimal equivalent automaton $\mathcal{A}_{\text{MIN}}$ can be constructed with the following two steps:*

(i) Applying weight pushing to $\mathcal{A}$ to obtain the pushed automaton $\mathcal{A}_{\text{PUSH}}$;

(ii) Unweighted minimization of $\mathcal{A}_{\text{PUSH}}$ treating the pairs of transition symbols and weights as single symbols.

Proof. Since $\mathcal{A}_{\text{MIN}}$ was obtained by merging *identical* states in $\mathcal{A}_{\text{PUSH}}$, Thm. 3.11.12 is equivalent to

$$\forall \boldsymbol{x}, \boldsymbol{y} \in P_L, \boldsymbol{x} R_L \boldsymbol{y} \implies n\left(\boldsymbol{\pi}_{q_{I\mathcal{A}_{\text{MIN}}},\boldsymbol{x}}\right) = n\left(\boldsymbol{\pi}_{q_{I\mathcal{A}_{\text{MIN}}},\boldsymbol{y}}\right). \tag{3.258}$$

Similarly to Thm. 3.11.11, this implies that the number of states is *less* than or equal to the number of equivalence classes of R_L. The only way this can occur is of course if the number of states is indeed *equal* to the number of equivalence classes, implying that the automaton $\mathcal{A}_{\text{MIN}}$ is indeed minimal. Since $\mathcal{A}_{\text{MIN}}$ is equivalent to $\mathcal{A}$, we conclude, by Thm. 3.11.11, that $\mathcal{A}_{\text{MIN}}$ is indeed a minimal WFSA accepting the same language as $\mathcal{A}$. ■

It can also easily be seen from Thm. 3.11.12 and Thm. 3.11.13 that the states constructed in the minimal automaton correspond to the equivalence classes of states in the original automaton—the equivalent states become identical after weight pushing and these are merged into blocks that form the states in the minimized machine.

As mentioned before, minimization of weighted automata is not unique due to the possible redistribution of the weights. However, the *topology* (the underlying unweighted graph) of all minimal automata equivalent to some starting automaton $\mathcal{A}$ is identical.

Theorem 3.11.14. *Let $\mathcal{A}$ be a WFSA. All minimal WFSA equivalent to $\mathcal{A}$ have the same topology and they only differ in the way the weights are distributed along the paths.*

Proof. As shown in Thm. 3.11.13, the application of weight pushing followed by unweighted minimization produces a minimal equivalent WFSA in which the states are the blocks of equivalent states in the original automaton. It is easy to see that any alternative partition of the states or definition of the transitions (modulo the weights) would lead to an non-equivalent FSA, meaning that the constructed minimal WFSA is unique up to the rearrangement of the weights. ■

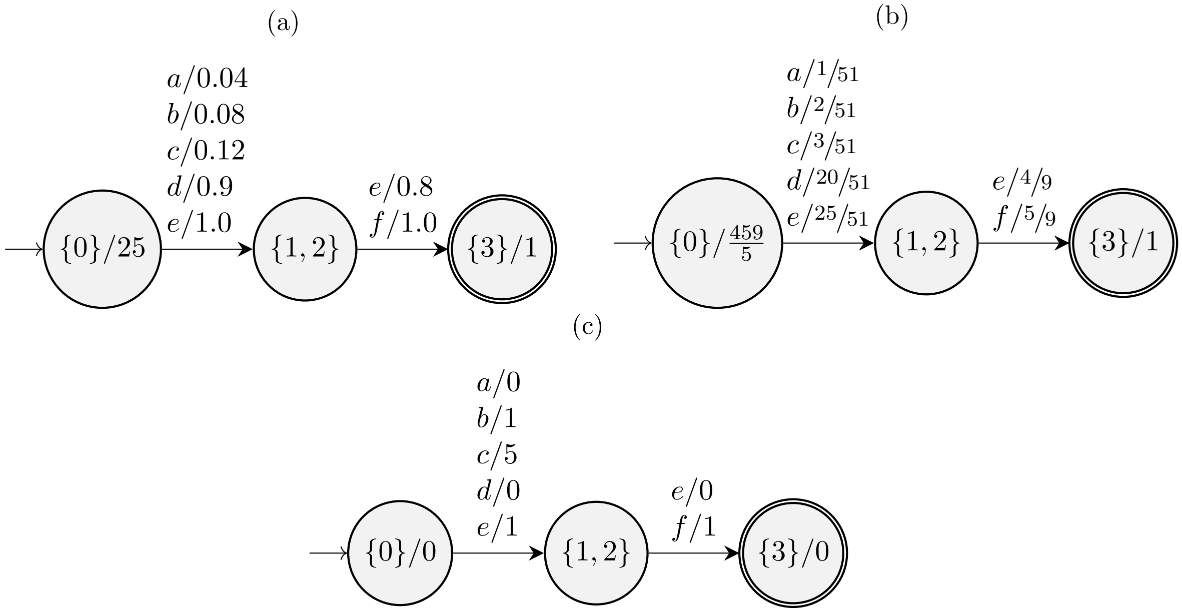


Figure 3.36: Fig. 3.36a and Fig. 3.36b show two minimal automata equivalent to the automaton $\mathcal{A}$ from Fig. 3.35a over the real semiring showing that minimization is not unique. Their topology, however, is identical. Fig. 3.36c shows an automaton equivalent to $\mathcal{A}$ over the Tropical semiring.

Example 3.11.7. *Fig. 3.36 show the result of the weighted minimization procedure applied to the automaton $\mathcal{A}$ from Fig. 3.35a over two different semirings. It also show that the minimization is not unique with regards to the weight distribution as we can have the equivalent minimal automata with different weights. Notice that the original automaton $\mathcal{A}$ is not suitable for minimization in its original form. We first perform weight pushing and minimize the automata as shown above and then minimize Fig. 3.35b or Fig. 3.35c, depending on the semiring.*

With this, we have shown that by pre-processing the input automaton $\mathcal{A}$ appropriately, we can reuse the unweighted minimization machinery to correctly minimize $\mathcal{A}$.

Complexity. The weighted minimization consists of the weight pushing and subsequent unweighted minimization, so its runtime is the sum of the two. We saw above that the runtime of weight pushing is in the general case $\mathcal{O}\left(|Q|^3\right)$. Since this is a first step of the weighted minimization, the runtime of it will be at least $\mathcal{O}\left(|Q|^3\right)$. Weight pushing does not alter the state space and as we saw, minimization can run in time $\mathcal{O}\left(|Q| \log\left(|Q|\right)\right)$. Therefore, the combined runtime is dominated by weight pushing and results in the bound $\mathcal{O}\left(|Q|^3\right)$.

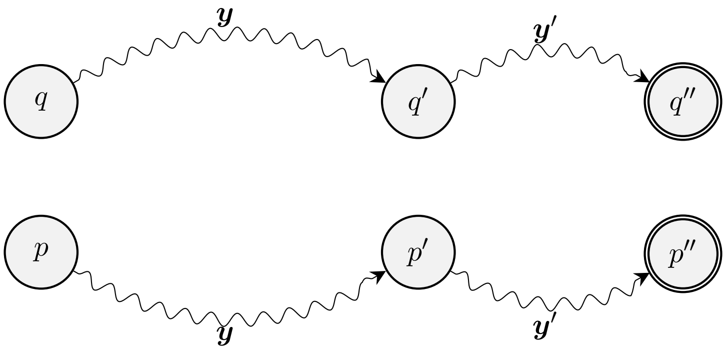


Figure 3.37: Paths yielding the suffix $\boldsymbol{y} \circ \boldsymbol{y}'$ starting from the equivalent states q, p. All pairs of intermediate states q', p' on the two paths are in the same equivalence class, i.e. $q' \sim_\delta p'$.

3.11.7 Faster Weight Pushing

It turns out we can accelerate weight pushing if, instead of using backward weights, we use a potential function that is faster to compute.

Recall that we used the backward weights $\boldsymbol{\beta}$ as the potential function. If weight pushing is used as a preliminary step before performing weighted minimization, this is a bottleneck. It turns out that we do not actually need the sum of the weights of all paths to the final states (backward weight) and we can do weight pushing using the backward weight for a single string. The resulting automaton does not become stochastic using this method but it still ensures that the automaton is canonical and weighted minimization can be done like in the unweighted case.

We start with an intuitive explanation of this faster algorithm. Recall that in the weighted case, the right languages of two states q and q', L and L', are *equivalent* if there exist scaling factors $k, k' \in \mathbb{K}$ such that

$$k^{-1} \otimes L = k'^{-1} \otimes L'. \tag{3.259}$$

The speedup is based on the following observation. The backward weights of any suffix $\boldsymbol{y}$ yielded by the paths starting from two equivalent states q and q' under $\mathcal{A}_{\text{PUSH}}$ must be equal, i.e. $\boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}(q, \boldsymbol{y}) = \boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}(q', \boldsymbol{y})$. Therefore the algorithm aims to find such a suffix $\boldsymbol{y}$ and its backward weights starting from two equivalent states q and q'. These weights are exactly the scaling factors that must get cancelled by the pushing algorithm. In order to do this, we need a pre-computed partition of the states $\mathcal{P}$ into equivalence classes. The algorithm proceeds in a breadth-first search manner starting from the final states until it computes a pushing weight for every co-accessible state. For every state in an equivalence class it computes a suffix $\boldsymbol{y}$ and its backward weight starting from this state. By exploring the states by their equivalence classes, we can ensure that the same string is assigned to every state in an equivalence class. Notice that all pairs of intermediate states on the paths yielding these suffixes must be also in the same equivalence class, as shown in Fig. 3.37. Alg. 31 formalizes this intuition.

Lemma 3.11.12. *Let q be some state in a WFSA $\mathcal{A}$ and $\boldsymbol{y}$ its suffix computed by the* Quasi-Backwards *algorithm. The potential weight $\Psi(q)$ computed by the WFSA* Quasi-Backwards *algorithm is equal to the backward weight $\boldsymbol{\beta}_{\mathcal{A}}(q, \boldsymbol{y})$.*

Proof. We prove the statement by induction on the length of the path yielding $\boldsymbol{y}$.

Base Case. Let $\boldsymbol{y}$ be the yield of the path $q \xrightarrow{a/w} p$ where $p \in F$. This path contains a single transition and its yield is the symbol a. As the automaton is deterministic, $\boldsymbol{\beta}_{\mathcal{A}}(q, a) = w \otimes \rho(q)$.

Algorithm 31 The WFSA Quasi-Backwards algorithm.

```
def Quasi-Backwards($\mathcal{A}$, $\mathcal{P}$):
  queue $\leftarrow \emptyset$
  $C \leftarrow F$                                         ▷All final states are trivially co-accessible.
  for $q \in F$ :
    $\Psi(q) \leftarrow \rho(q)$                           ▷Potentials of final states are final weights.
    $\boldsymbol{y}(q) \leftarrow \varepsilon$             ▷Assign empty string $\varepsilon$ to final states.
    for $p \xrightarrow{\bullet/\bullet} q \in \delta$ :
      push transition $p \xrightarrow{\bullet/\bullet} q$ onto queue    ▷Add to queue all transitions to final states.
  while |queue| $> 0$ :
    pop $p \xrightarrow{a/\bullet} q$ from queue
    if $p \notin C$ :
      $C \leftarrow C \cup [p]$                            ▷Set equivalence class of $p$ as co-accessible.
      for $r \in [p]$ :
        ▷This loop only iterates once as $\mathcal{A}$ is deterministic.
        for $r \xrightarrow{a/w} s$ in $\delta$ :
          $\Psi(r) \leftarrow w \otimes \Psi(s)$
          ▷All states in $p$'s equivalence class get assigned the same string as all states $s$ are also in the same
           equivalence class.
          $\boldsymbol{y}(r) \leftarrow a \circ \boldsymbol{y}(s)$
        for $\bullet \xrightarrow{\bullet/\bullet} r \in \delta$ :
          push transition $\bullet \xrightarrow{\bullet/\bullet} r$ onto queue
  return $\Psi$, $\boldsymbol{y}$
```

As p's potential weight is initialized to $\rho(q)$, it holds that

$$\begin{aligned}\boldsymbol{\beta}_{\mathcal{A}}(q,a) &= w \otimes \rho(q) && (3.260)\\ &= w \otimes \Psi(p) && (3.261)\\ &= \Psi(q)\,. && (3.262)\end{aligned}$$

Inductive Step. Let $\left(q \xrightarrow{a/w} p, p \xrightarrow{\bullet/\bullet} \bullet \cdots\right)$ be a path of length n from the state q to a final state. Assume moreover that its yield is $a \circ \boldsymbol{y}$, where $\boldsymbol{y} \in \Sigma^*$ is a suffix yielded by the path starting from p. By the inductive hypothesis, $\Psi(p) = \boldsymbol{\beta}_{\mathcal{A}}(p, \boldsymbol{y})$. As the automaton is deterministic, there is a single transition from q to p that is labeled with a. Therefore we get the following:

$$\begin{aligned}\boldsymbol{\beta}_{\mathcal{A}}(q, a \circ \boldsymbol{y}) &= w \otimes \boldsymbol{\beta}_{\mathcal{A}}(p, \boldsymbol{y}) && (3.263)\\ &= w \otimes \Psi(p) && (3.264)\\ &= \Psi(q)\,. && (3.265)\end{aligned}$$

This proves the correctness of the algorithm. ■

Complexity. At initialization we set the potential weights and suffixes of the final states. This initialization takes time $\mathcal{O}(|F|) \subseteq \mathcal{O}(|Q|)$. Additionally, all transitions to the final states are added to queue which takes $\mathcal{O}(|\delta|)$. Inside the main loop, each transition can be pushed then popped at most once. This manipulation thus takes $\mathcal{O}(|\delta|)$. If the source state of the current transition is not marked as co-accessible yet, its potential weight and suffix need to be computed. This is done once per state, thus it takes $\mathcal{O}(|Q|)$. In total we get a complexity of $\mathcal{O}(|Q| + |\delta|)$, which is a lot faster than the $\mathcal{O}(|Q|^3)$ complexity of the backward algorithm.

Lemma 3.11.13. *Let $\mathcal{A}$ be a WFSA and $p{\sim_\delta}p'$ and $q{\sim_\delta}q'$ two pairs of equivalent states in $\mathcal{A}$. After pushing it holds that the transitions $q \xrightarrow{a/w_{\mathcal{A}_{\text{PUSH}}}} p$ and $q' \xrightarrow{a/w_{\mathcal{A}_{\text{PUSH}}}} p'$ labeled with the same symbol have the same weight $w_{\mathcal{A}_{\text{PUSH}}}$.*

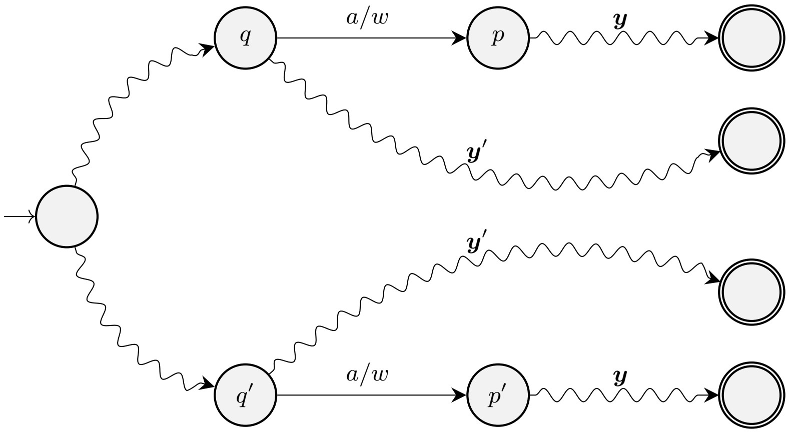


Figure 3.38: Example of corresponding edges $q \xrightarrow{a/w} p$ and $q' \xrightarrow{a/w} p'$ for the equivalent states $p{\sim_\delta}p'$ and $q{\sim_\delta}q'$. The pushing weights of states p and p' are $\boldsymbol{\beta}_{\mathcal{A}}(p, \boldsymbol{y})$ and $\boldsymbol{\beta}_{\mathcal{A}}(p', \boldsymbol{y})$, respectively. Additionally, the pushing weights of states p and p' are $\boldsymbol{\beta}_{\mathcal{A}}(q, \boldsymbol{y}')$ and $\boldsymbol{\beta}_{\mathcal{A}}(q', \boldsymbol{y}')$. Notice that p and p' do *not* necessarily need to be on the paths yielding $\boldsymbol{y}'$.

Lemma 3.11.14. *Let $\mathcal{A} = (\Sigma, Q, I_1, F_1, \delta_1, \lambda_1, \rho_1)$ be a deterministic WFSA and $\mathcal{A}_{\text{PUSH}} = (\Sigma, Q, I_2, F_2, \delta_2, \lambda_2, \rho_2)$ be its pushed variant. Then for any two states $q, q' \in Q$, $[q] = [q']$ (q being equivalent to q') implies the following:*

$$\forall \boldsymbol{y} \in \Sigma^* : n\left(\boldsymbol{\pi}\left(q, \boldsymbol{y}\right)\right) \in F \implies \boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}\left(q, \boldsymbol{y}\right) = \boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}\left(q', \boldsymbol{y}\right). \tag{3.266}$$

Proof. If $\mathcal{A}$ is deterministic, for any string $\boldsymbol{y}$, the backward weight $\boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}\left(q, \boldsymbol{y}\right)$ is the weight of the unique path yielding $\boldsymbol{y}$. Let $\boldsymbol{y} = s_1 \cdots s_n$ be a string of length $|\boldsymbol{y}| = n$. Additionally, let $\boldsymbol{\pi}_{q,\boldsymbol{y}} = \left(q_0 = q \xrightarrow{s_1/w_1} q_1 \cdots q_{n-1} \xrightarrow{s_n/w_n} q_n\right)$ and $\boldsymbol{\pi}_{q',\boldsymbol{y}} = \left(q'_0 = q' \xrightarrow{s_1/w'_1} q'_1 \cdots q'_{n-1} \xrightarrow{s_n/w'_n} q'_n\right)$ be the paths yielding $\boldsymbol{y}$ starting from q and q' respectively. We have just proven in Lemma 3.11.13 that all pairs of edges $q_{i-1} \xrightarrow{s_i/w_i} q_i$ and $q'_{i-1} \xrightarrow{s_i/w_i} q'_i$ are weighted equally after pushing. Then the following must hold:

$$\boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}\left(q, \boldsymbol{y}\right) = w_1 \otimes w_2 \otimes \cdots \otimes w_n = \boldsymbol{\beta}_{\mathcal{A}_{\text{PUSH}}}\left(q', \boldsymbol{y}\right) \tag{3.267}$$

This concludes the proof. ■

Theorem 3.11.15. *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)_1$ be a deterministic WFSA and $q, q' \in Q$. If q and q' are* equivalent*, their right language is* identical *after weight pushing.*

Proof. This is identical to Thm. 3.11.12 and can be proven using the same arguments. ■

This result implies that weighted minimization on a WFSA can be done as described in the following theorem.

Theorem 3.11.16. *Let $\mathcal{A}$ be a WFSA. An equivalent minimal automaton $\mathcal{A}_{\text{MIN}}$ be computed using the following three steps:*

(i) Computing the coarsest partition $\mathcal{P}$ of the states on the unweighted version of $\mathcal{A}$;

(ii) Weight pushing using the Quasi-Backwards algorithm to compute the pushing weights;

(iii) Unweighted minimization by refining the partition $\mathcal{P}$ on the automaton in which the symbols and weights are treated as a single label.

Proof. First notice that any state q' that is not equivalent to q in the underlying unweighted automaton cannot be equivalent in $\mathcal{A}$ as $\mathcal{P}$ is the coarsest partition of the states. Additionally, we have just proven in Lemma 3.11.13 that any pair of equivalent states $\mathcal{A}$ have the same edge weights after pushing. This implies that equivalent states must have the same labels (a, w) on the outgoing transitions in $\widetilde{\mathcal{A}_{\text{PUSH}}}$. This allows us to do the refinement step of $\mathcal{P}$ in the last step of the algorithm. ■

Complexity. Computing the initial partition $\mathcal{P}$, as well as the refinement from the last step can be done in time $\mathcal{O}(|Q|\log(|Q|))$. Additionally, computing the pushing weights has a runtime of $\mathcal{O}(|Q| + |\delta|)$. Lastly, constructing the minimal automaton runs in linear time. In total we get a complexity of $\mathcal{O}(|Q|\log(|Q|) + |\delta|)$.

3.11.8 NFA Equivalence Testing

So far, we have looked at equivalence and minimization in the context of *deterministic* automata, so in order to minimize or compare nondeterministic automata, we had to perform determinization as a preprocessing step. However, determinization has exponential time complexity in the worst case, provided the automaton is even determinizable at all! We now introduce algorithms to do equivalence testing and minimization *directly on nondeterministic automata*.

To do this, we will slightly change our perspective and look at WFSA through the lens of linear algebra, following closely the summary and notation provided by Kiefer (2020). We will start by introducing some further algebraic concepts that will be used in this section. For starters, we will require *semiringset* to be a *field*, which is defined as follows:

Definition 3.11.13 (Field)**.** *A **field** $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ is a* semifield *as defined in Def. 3.10.2 with the following additional axioms:*

- *All elements $x \in \mathbb{K}$ admit an additive inverse $-x \in \mathbb{K}$, such that $x \oplus -x = \boldsymbol{0}$;*
- *$(\mathbb{K}, \otimes)$ is an **abelian** group, meaning $\otimes$ is commutative: $\forall x, y \in \mathbb{K},\ x \otimes y = y \otimes x$;*

We also define the concept of a *vector space* over a field as follows.

Definition 3.11.14 (Vector space)**.** *A **vector space** over a field $\mathcal{W}$ is a set $\mathbb{V}$ together with two binary operations $(+, \times)$, called vector addition and vector multiplication, satisfying the following axioms:*

1. ***Associativity** of vector addition: $\forall \mathbf{v}, \mathbf{u}, \mathbf{q} \in \mathbb{V}$*
$$(\mathbf{v} + \mathbf{u}) + \mathbf{q} = \mathbf{v} + (\mathbf{u} + \mathbf{q}) \tag{3.268}$$
2. ***Commutativity** of vector addition: $\forall \mathbf{v}, \mathbf{u} \in \mathbb{V}$*
$$\mathbf{v} + \mathbf{u} = \mathbf{u} + \mathbf{v} \tag{3.269}$$
3. ***Identity** element of vector addition: there exists $\boldsymbol{0} \in \mathbb{V}$ such that $\forall \mathbf{v} \in \mathbb{V}$*
$$\mathbf{v} + \boldsymbol{0} = \mathbf{v} \tag{3.270}$$
4. ***Inverse** elements of vector addition: for every $\mathbf{v} \in \mathbb{V}$ there exists a $-\mathbf{v} \in \mathbb{V}$ such that*
$$\mathbf{v} + (-\mathbf{v}) = \boldsymbol{0} \tag{3.271}$$
5. ***Compatibility** of scalar multiplication with field multiplication: $\forall \mathbf{v} \in \mathbb{V}$ and $x, y \in \mathcal{W}$*
$$x\,(y\mathbf{v}) = (xy)\,\mathbf{v} \tag{3.272}$$
6. ***Identity** element of scalar multiplication: $\forall \mathbf{v} \in \mathbb{V}$*
$$\boldsymbol{1}\mathbf{v} = \mathbf{v} \tag{3.273}$$
7. ***Distributivity** of scalar multiplication with respect to vector addition: $\forall x \in \mathcal{W}, \forall \mathbf{u}, \mathbf{v} \in \mathbb{V}$*
$$x\,(\mathbf{v} + \mathbf{u}) = x\mathbf{v} + x\mathbf{u} \tag{3.274}$$

8. ***Distributivity*** *of scalar multiplication with respect to field addition:* $\forall x, y \in \mathcal{W}, \forall \mathbf{v} \in \mathbb{V}$

$$(x + y)\,\mathbf{v} = x\mathbf{v} + y\mathbf{v} \tag{3.275}$$

We also define the notions of span, basis, and rank:

Definition 3.11.15 (Span)**.** *Let* $\mathbb{V}$ *be a vector space over field* $\mathcal{W}$*. Let* $V = \{\mathbf{v}_1 \dots \mathbf{v}_k\}$ *be a set of vectors in* $\mathbb{V}$*. Then the* ***span*** *of* V *is defined as the set of linear combinations of the vectors in* V*:*

$$span(V) \overset{\text{def}}{=} \left\{ \bigoplus_{i=1}^{k} w_i \mathbf{v}_i \mid \mathbf{v}_i \in V, w_i \in \mathcal{W} \right\} \tag{3.276}$$

In the following, we will write the span of V as $\langle \mathbf{v} \mid \mathbf{v} \in V \rangle$.

Definition 3.11.16 (Basis)**.** *Let* $\mathbb{V}$ *be a vector space over field* $\mathcal{W}$*. Then* $B = \{\mathbf{b}_1 \dots \mathbf{b}_k\}$ *is a* ***basis*** *of* $\mathbb{V}$ *if (1)* B *spans* $\mathbb{V}$*, i.e.* $\mathbb{V} = span(B)$*, and (2) all vectors in* B *are linearly independent, i.e.:*

$$\bigoplus_{i=1}^{k} w_i \mathbf{b}_i = \mathbf{0} \implies w_1 = \cdots = w_k = \mathbf{0} \tag{3.277}$$

Definition 3.11.17 (Rank)**.** *Let* $\mathbf{M}$ *be a matrix. Then the* ***rank*** *of* $\mathbf{M}$ *is equal to the number of* linearly independent *columns of* $\mathbf{M}$.

Definition 3.11.18 (Orthogonality)**.** *Let* $\mathbb{V}$ *be a vector space over field* $\mathcal{W}$*. Two vectors* $\mathbf{v}, \mathbf{v}'$ *are called* ***orthogonal*** *if* $\mathbf{v}^\top \mathbf{v}' = \mathbf{v}'^\top \mathbf{v} = \mathbf{0}$*. We write* $\mathbf{v} \perp \mathbf{v}'$.

Definition 3.11.19 (Projection)**.** *Let* $\mathbb{V}$ *be a vector space over field* $\mathcal{W}$*. Let* $\mathbf{u}, \mathbf{v} \in \mathbb{V}$ *be vectors in* $\mathbb{V}$*. We define the* ***projection*** *of* $\mathbf{v}$ *onto* $\mathbf{u}$ *as:*

$$proj_{\mathbf{u}}(\mathbf{v}) = \frac{\mathbf{v}^\top \mathbf{u}}{\mathbf{u}^\top \mathbf{u}} \mathbf{u} \tag{3.278}$$

Gram-Schmidt process. A very useful method in the context of vectors and bases is the Gram-Schmidt orthogonalization process. It allows us to convert any set of vectors into a (potentially smaller) set of orthogonal vectors. This works as follows: Take a set of vectors $V = \{\mathbf{v}_1, \dots \mathbf{v}_n\}$. We want to find another set of vectors $U = \{\mathbf{u}_1, \dots, \mathbf{u}_m\}$ such that $span(U) = span(V)$, with the condition that $\forall \mathbf{u}, \mathbf{u}' \in U, \mathbf{u} \neq \mathbf{u}' : \mathbf{u} \perp \mathbf{u}'$. We do this in the following way:

1. **Base case**: $U = \{\emptyset\}$. Add $\mathbf{u}_1 = \mathbf{v}_1$ to U for some $\mathbf{v}_1 \in V$.
2. **For** $k = 2 \dots n$: $\mathbf{u} = \mathbf{v}_k - \bigoplus_{j=1}^{k-1} proj_{\mathbf{u}_j}(\mathbf{v}_k)$. If $\mathbf{u} \neq \mathbf{0}$, add it to U.

Intuitively, at ever step after the base case, for each new vector $\mathbf{v}$, we remove all the components of it that can be accounted for by the vectors already in U. The remaining vector is then orthogonal to the other vectors in U and is added to U. The second step can also be used to check if a vector is in the span of an orthogonal basis: If after subtracting all the projections onto the basis, the remaining vector is $\mathbf{0}$, the vector is in the span.

We can now start thinking about an ε-free WFSA $\mathcal{A}(\Sigma, Q, \delta, \lambda, \rho)$ purely in terms of linear algebra:

- Instead of having an explicit set of states Q, let $n = |Q| \in \mathbb{N}$ be the number of states of the automaton and have the natural numbers up to n represent an (arbitrarily) ordered list of states $\{q_1, q_2, \dots, q_n\}$;

- Let $\mathbf{M} : \Sigma \rightarrow \mathcal{W}^{n \times n}$ be a linear map from symbols to state-pair transition weights. We then have a one-to-one correspondence with the transition weight function δ and $\mathbf{M}$, where $\forall a \in \Sigma : \ \mathbf{M}(a)_{ij} = \delta(q_i, a, q_j)$;
- We extend the transition matrix $\mathbf{M} : \Sigma^* \rightarrow \mathcal{W}^{n \times n}$ to be a map from sequences to weights, where $\mathbf{M}(\varepsilon) \overset{\text{def}}{=} \mathbf{I}_n$ and $\mathbf{M}(a_1 \dots a_k) \overset{\text{def}}{=} \mathbf{M}(a_k) \dots \mathbf{M}(a_k)$;
- Let $\boldsymbol{\lambda} \in \mathcal{W}^n$ be a vector of initial weights, where each value $\boldsymbol{\lambda}_i$ corresponds to the initial weight of the state q_i, i.e. $\lambda\,(q_i)$;
- Let $\boldsymbol{\rho} \in \mathcal{W}^n$ be a vector of final weights, where each value $\boldsymbol{\rho}_i$ corresponds to the final weight of the state q_i, i.e. $\rho\,(q_i)$.

Now if we have an arbitrary ε-free WFSA $\mathcal{A}$ whose formal power series is $r = \sum_{\boldsymbol{y} \in \Sigma^*} r(\boldsymbol{y})\boldsymbol{y}$, we can simply get the coefficients $r(\boldsymbol{y})$ (the weights of a strings $\boldsymbol{y}$ under $\mathcal{A}$) by calculating $r(\boldsymbol{y}) = \boldsymbol{\lambda}^\top \mathbf{M}(\boldsymbol{y})\boldsymbol{\rho}$.

Definition 3.11.20 (WFSA Equivalence)**.** *Let $\mathcal{A}_1, \mathcal{A}_2$ be two WFSA with* formal power series *r_1 and r_2, respectively. We say $\mathcal{A}_1$ and $\mathcal{A}_2$ are **equivalent automata** if their formal power series are the same, i.e. if $r_1 = r_2$.*

We have already seen in §2.3 that formal power series are closed under certain operations, such as addition.

Definition 3.11.21 (Weighted union)**.** *Let $\mathcal{A}_1, \mathcal{A}_2$ be WFSA. We now define the **union automaton** $\mathcal{A}_+$ of $\mathcal{A}_1$ and $\mathcal{A}_2$ as the automaton whose power series, r_+, is the sum of the power series of the other two, i.e. $r_+ = r_1 + r_2$.*

Definition 3.11.22 (Weighted difference)**.** *Similarly, let the **difference automaton** $\mathcal{A}_-$ of $\mathcal{A}_1$ and $\mathcal{A}_2$ be the automaton whose power series, r_-, is the difference between the power series of the two automata, i.e. $r_- = r_1 - r_2$.*

Using the above linear algebra representation for WFSA introduced above, we can compute this efficiently, as the following proposition shows.

Proposition 3.11.1. *Let $\mathcal{A}_1, \mathcal{A}_2$ be WFSA with $n_i, \mathbf{M}_i, \mathbf{M}_i, \boldsymbol{\lambda}_i, \boldsymbol{\rho}_i$ for $i \in \{1, 2\}$ as defined above. Then computing the union and difference automata takes $\mathcal{O}(|\Sigma|(n_1 + n_2)^2)$ time, i.e. it is quadratic in the total number of states of the two automata.*

Proof. Let $\mathcal{A}_+$ be the automaton with transition function and initial and final weights described by:

$$\forall a \in \Sigma : \mathbf{M}_+(a) = \begin{pmatrix} \mathbf{M}_1(a) & 0_{n_1 \times n_2} \\ 0_{n_2 \times n_1} & \mathbf{M}_2(a) \end{pmatrix} \qquad \boldsymbol{\lambda}_+^\top = (\boldsymbol{\lambda}_1^\top, \boldsymbol{\lambda}_2^\top) \qquad \boldsymbol{\rho}_+ = \begin{pmatrix} \boldsymbol{\rho}_1 \\ \boldsymbol{\rho}_2 \end{pmatrix} \tag{3.279}$$

Then the coefficients for strings $\boldsymbol{y} = a_1 \dots a_k$ of the power series r_+ of $\mathcal{A}_+$ are given by:

$$\begin{aligned} r(\boldsymbol{y})_+ &= \boldsymbol{\lambda}_+^\top \mathbf{M}(\boldsymbol{y})_+ \boldsymbol{\rho}_+ && (3.280) \\ &= \boldsymbol{\lambda}_+^\top \mathbf{M}(a_1)_+ \dots \mathbf{M}(a_k)_+ \boldsymbol{\rho}_+ && (3.281) \\ &= \boldsymbol{\lambda}_+^\top \begin{pmatrix} \mathbf{M}_1(a_1) \dots \mathbf{M}_1(a_k) & 0_{n_1 \times n_2} \\ 0_{n_2 \times n_1} & \mathbf{M}_2(a_1) \dots \mathbf{M}_2(a_k) \end{pmatrix} && (3.282) \\ &= (\boldsymbol{\lambda}_1^\top, \boldsymbol{\lambda}_2^\top) \begin{pmatrix} \mathbf{M}(\boldsymbol{y})_1 & 0_{n_1 \times n_2} \\ 0_{n_2 \times n_1} & \mathbf{M}(\boldsymbol{y})_2 \end{pmatrix} \begin{pmatrix} \boldsymbol{\rho}_1 \\ \boldsymbol{\rho}_2 \end{pmatrix} && (3.283) \\ &= \boldsymbol{\lambda}_1^\top \mathbf{M}(\boldsymbol{y})_1 \boldsymbol{\rho}_1 + \boldsymbol{\lambda}_2^\top \mathbf{M}(\boldsymbol{y})_2 \boldsymbol{\rho}_2 && (3.284) \\ &= r(\boldsymbol{y})_1 + r(\boldsymbol{y})_2 && (3.285) \end{aligned}$$

Note that we can construct $\mathcal{A}_-$ in exactly the same way except changing e.g. the new vector of initial weights to $\boldsymbol{\lambda}_-^\top = (\boldsymbol{\lambda}_1^\top, -\boldsymbol{\lambda}_2^\top)$ to get $r(\boldsymbol{y})_- = r(\boldsymbol{y})_1 - r(\boldsymbol{y})_2$. Clearly, the new transition matrix in both cases has dimension $(n_1 + n_2)^2$, and since we have one matrix for each symbol, this results in the desired complexity, $\mathcal{O}(|\Sigma|(n_1 + n_2)^2)$. ■

In the following, we will denote the construction of $\mathcal{A}_+$ and $\mathcal{A}_-$ as described in the proof above by NFAUnion and NFADifference, respectively.

Fliess' Theorem

In order to be able to test equivalence and find the minimal automaton using the representation introduced above, we will make use of *Fliess' Theorem* (Fliess, 1974). This theorem has two parts; the first part, introduced here, is relevant for equivalence, while the second part concerns minimization and will be discussed in the next section.

First, we first introduce the notion of a *Hankel matrix* in the context of formal languages.

Definition 3.11.23. *Let $r \in \Sigma^* \to \mathcal{W}$ be the power series of a weighted language L. Then the* ***Hankel matrix*** *of r is the (infinite) matrix $H_r \in \mathcal{W}^{\Sigma^* \times \Sigma^*}$ defined by*

$$H_r(\boldsymbol{p}, \boldsymbol{s}) = r(\boldsymbol{ps}) \tag{3.286}$$

We call $\boldsymbol{p}$ the ***prefix*** *and $\boldsymbol{s}$ the* ***suffix*** *of the string $\boldsymbol{y} = \boldsymbol{ps}$. The prefix indexes the rows of the Hankel matrix, and the suffix indexes the columns.*

This means we can describe a weighted formal language by constructing its Hankel matrix, albeit generally an infinite one. Intuitively it makes sense that the weight of a string is independent of how it is decomposed, and hence, there is naturally some duplication in the Hankel matrix (by design). Specifically, we have that two different decompositions of the same string lead to the same weight:

$$\forall \boldsymbol{p}, \boldsymbol{p}', \boldsymbol{s}, \boldsymbol{s}' \in \Sigma^* : \boldsymbol{ps} = \boldsymbol{p}'\boldsymbol{s}' \implies H_r(\boldsymbol{p}, \boldsymbol{s}) = H_r(\boldsymbol{p}', \boldsymbol{s}') \tag{3.287}$$

Example 3.11.8. *Take the alphabet $\Sigma = \{a, b\}$ and some weighted language L over Σ with formal power series r. Then the* Hankel matrix *of r is:*

$$\begin{array}{c} \\ \varepsilon \\ a \\ b \\ aa \\ ab \\ \vdots \end{array}
\begin{array}{c} \begin{array}{cccccc} \varepsilon & a & b & aa & ab & \cdots \end{array} \\
\begin{pmatrix} r(\varepsilon) & r(a) & r(b) & r(aa) & r(ab) & \\ r(a) & r(aa) & r(ab) & r(aaa) & r(aab) & \\ r(b) & r(ba) & r(bb) & r(baa) & r(bab) & \\ r(aa) & r(aaa) & r(aab) & r(aaaa) & r(aaab) & \cdots \\ r(ab) & r(aba) & r(abb) & r(abaa) & r(abab) & \\ & & & \vdots & & \end{pmatrix} \end{array}$$

We also define two additional vector spaces, the *forward space*, and the *backward space*:

Definition 3.11.24 (Forward space)**.** *We define the* ***forward space*** *$\mathcal{F}$ of an automaton $\mathcal{A}$ as the vector space spanned by row vectors of sequence prefix weights, where the ith index of a vector corresponds to the cumulative weight of paths over string str* ending *in q_i:*

$$\mathcal{F} \overset{\text{def}}{=} \langle \boldsymbol{\lambda}^\top \mathbf{M}(\boldsymbol{y}) \mid \boldsymbol{y} \in \Sigma^* \rangle \tag{3.288}$$

Definition 3.11.25 (Backward space). *Conversely, the* ***backward space*** $\mathcal{B}$ *of automaton* $\mathcal{A}$ *as the vector space spanned by column vectors of sequence suffix weights, where the ith index of a vector corresponds to the cumulative weight of paths over string str* starting *in* q_i*:*

$$\mathcal{B} \overset{\text{def}}{=} \langle \mathbf{M}(\boldsymbol{y})\boldsymbol{\rho} \mid \boldsymbol{y} \in \Sigma^* \rangle \tag{3.289}$$

Now, we can decompose any entry in the Hankel matrix as the product of a vector of the forward space and a vector of the backward space, as the following proposition shows:

Proposition 3.11.2. *Let* $\boldsymbol{y} = \boldsymbol{ps}$ *be a string composed of prefix* $\boldsymbol{p}$ *and suffix* $\boldsymbol{s}$*. Then the weight of* $\boldsymbol{y}$ *can be decomposed as follows:*

$$H_r(\boldsymbol{p}, \boldsymbol{s}) = \boldsymbol{\lambda}^\top \mathbf{M}(\boldsymbol{p}) \times \mathbf{M}(\boldsymbol{s})\boldsymbol{\rho} \tag{3.290}$$

Proof. This follows directly from the definition of the Hankel matrix and the string weights:

$$\boldsymbol{\lambda}^\top \mathbf{M}(\boldsymbol{p}) \times \mathbf{M}(\boldsymbol{s})\boldsymbol{\rho} = \boldsymbol{\lambda}^\top \mathbf{M}(\boldsymbol{ps})\boldsymbol{\rho} = \boldsymbol{\lambda}^\top \mathbf{M}(\boldsymbol{y})\boldsymbol{\rho} = r(\boldsymbol{y}) = r(\boldsymbol{ps}) = H_r(\boldsymbol{p}, \boldsymbol{s}) \tag{3.291}$$

■

Now the first part of Fliess' theorem concerns the correspondences between an automaton and the Hankel matrix of the weighted language it recognizes:

Theorem 3.11.17 (Fliess 1). *Let* $\mathcal{A}$ *be an* ε*-free WFSA with* n *states that recognizes the weighted language* L *with power series* r*, and let* H_r *be that language's Hankel matrix. Then* $\mathcal{A}$ *has at least* $rank(H_r)$ *states.*

Proof. Let F_r denote the $\mathcal{W}^{\Sigma^* \times n}$ matrix whose rows are the vectors of $\mathcal{F}$. Let further B_r denote the $\mathcal{W}^{n \times \Sigma^*}$ matrix whose columns consist of the vectors of $\mathcal{B}$. Then by Prop. 3.11.2, we can write the Hankel matrix of $\mathcal{A}$ as $H_r = F_r \times B_r$. Since F_r has n rows, and B_r has n columns, it follows that their rank is at most n. Due to the rank constraint of matrix multiplication, it follows that $rank(H_r) \leq n$. ■

With the decomposition into forward and backward space, we can now test the equivalence of two automata: A straightforward way to do this is to compute their *difference automaton*, and then check whether that difference is *zero*. If it is, then the two automata are equivalent! To do this, we first need to define what it means for an automaton to be *zero*:

Definition 3.11.26 (Zero automaton). *An automaton* $\mathcal{A}$ *is called* ***zero****, if the weight of any string in the automaton is zero, i.e.* $\forall \boldsymbol{y} \in \Sigma^* : r(\boldsymbol{y}) = \boldsymbol{\lambda}^\top \mathbf{M}(\boldsymbol{y})\boldsymbol{\rho} = 0$.

Note that an automaton is zero exactly when its forward space and its backward space are orthogonal, since then the string weight of any string will be zero. It is easy to see that this corresponds to either the forward space $\mathcal{F}$ being orthogonal to $\boldsymbol{\rho}$, or, equivalently, the backward space $\mathcal{B}$ being orthogonal to $\boldsymbol{\lambda}$. We hence have a very quick way of testing whether an automaton is zero:

Proposition 3.11.3. *Let* $\mathcal{A}$ *be a WFSA and* $\mathcal{F}$ *its forward space. Given a basis* F *of* $\mathcal{F}$*, we can test whether* $\mathcal{A}$ *is* zero *in polynomial time* ($\mathcal{O}(n^2)$)*.*

Proof. We can see from Def. 3.11.26 that for an automaton to be zero it suffices to show that the final weight vector $\boldsymbol{\rho}$ is orthogonal to the forward space $\mathcal{F}$. If we have a basis F of $\mathcal{F}$, we can hence find if this is the case in $|F|$ operations. As we saw in the proof of Thm. 3.11.17, the dimension of $\mathcal{F}$ is at most n. We can hence test whether $\mathcal{A}$ is *zero* by calculating $\mathbf{b}\boldsymbol{\rho}\ \forall \mathbf{b} \in F$, which takes $\mathcal{O}(n^2)$ time. ■

So, how do we get such a basis of the forward space? A first attempt at this is shown in Alg. 32. The algorithm proceeds as follows: It first checks whether the initial weight vector is $\mathbf{0}$ in line 2, since, if that is the case, the forward space will be $\mathbf{0}$ as well. We then start iteratively building generating a basis from a *basis vocabulary* V, that is, a set of strings $\boldsymbol{y}$ which have linearly independent forward (i.e. prefix weight) vectors $\boldsymbol{\lambda}^\top \mathbf{M}(\boldsymbol{y})$. At each iteration, we iterate through all words found so far (line 6), and then append them with every possible symbol a (lines 7-8) to check whether the prefix weight of this new *candidate sequence* is in the vector space spanned by the current basis (line 11). If it is not, we add it (line 12). We keep doing this until no more words can be found whose prefix weights are not in the basis, at which point the algorithm terminates.

Algorithm 32 Computing a basis of the forward space of an automaton.

```
1.  def ForwardBasis($\mathcal{A} = (\Sigma, n, \mathbf{M}, \boldsymbol{\lambda}, \boldsymbol{\rho})$):
2.    if $\boldsymbol{\lambda} = \mathbf{0}$ :
3.      return $\emptyset$
4.    $V \leftarrow \{\varepsilon\}$                          ▷Initialize the basis vocabulary with the empty string
5.    while True :
6.      for $\mathbf{w} \in V$ :
7.        for $a \in \Sigma$ :
8.          $\mathbf{w}' \leftarrow \mathbf{w}a$                  ▷Create new sequence by extending with a single symbol
9.          $\mathbf{b}' \leftarrow \boldsymbol{\lambda}^\top \mathbf{M}(\mathbf{w}')$            ▷Compute new candidate basis vector from sequence
10.         $F \leftarrow \{\boldsymbol{\lambda}^\top \mathbf{M}(\boldsymbol{y}) \mid \boldsymbol{y} \in V\}$          ▷Compute basis from vocabulary
11.         if $\mathbf{b}' \notin \langle \mathbf{b} \mid \mathbf{b} \in F \rangle$ :   ▷Check if the new sequence's prefix weight vector is not in the basis span
12.           $V \leftarrow V \cup \{\mathbf{w}'\}$              ▷If not, add it to the basis vocabulary
13.     if no change was made to $V$ in this iteration :
14.       return $F$
```

As can be easily seen, Alg. 32 is not a very efficient method for computing a basis of the forward space. This is, on the one hand, because we recompute the basis vectors of all the vocabulary sequences in line 10 in each iteration of the innermost for loop. Another reason is that we re-check the same words over and over again. So, to speed up the algorithm we can improve it twofold: Firstly, we store the vectors $\boldsymbol{\lambda}^\top \mathbf{M}(\boldsymbol{y})$ directly to avoid recomputing them. Secondly, we add each newly found sequence to a queue, so it only contributes to the basis vector computation once. The improved algorithm is shown in Alg. 33.

Proposition 3.11.4. *The improved Alg. 33 computes a basis of the forward space $\mathcal{F}$ of an automaton $\mathcal{A}$ in polynomial time ($\mathcal{O}(|\Sigma| n^3)$).*

Proof. As mentioned in Prop. 3.11.3, the dimension of $\mathcal{F}$, which we shall denote by $\overrightarrow{n}$, is less or equal to n, meaning that its basis has $\overrightarrow{n} \leq n$ vectors. Hence, the while loop in lines 6-14 will be executed at most $\mathcal{O}(n)$ times. In each iteration, the for loop in lines 8-13 iterates through the alphabet, adding a factor of $\mathcal{O}(|\Sigma|)$. Finally, we have the check in line 11 which tests whether the candidate basis vector $\mathbf{b}'$ is linearly independent of the current basis. This can be done in $\mathcal{O}(n^2)$ by using the Gram-Schmidt process to keep the basis in orthogonal form. This yields the overall complexity of $\mathcal{O}(|\Sigma| n^3)$. ■

We now have everything we need to test automata for equivalence. Alg. 34 combines the parts from above in an algorithm that takes two ε-free WFSA and returns *True* if they are equivalent and *False* otherwise. It does this by first computing the difference automaton $\mathcal{A}_-$ between the two automata using NFADifference, then obtaining a basis of its forward space using Alg. 33, and finally

Algorithm 33 Computing a basis of the forward space of an automaton efficiently.

```
def ForwardBasisModified($\mathcal{A} = (\Sigma, n, \mathbf{M}, \boldsymbol{\lambda}, \boldsymbol{\rho})$):
  if $\boldsymbol{\lambda} = \mathbf{0}$ :
    return $\emptyset$
  $F \leftarrow \{\boldsymbol{\lambda}^\top\}$                ▷Initialize the basis with the prefix weight vector of the empty string
  $Q \leftarrow [\boldsymbol{\lambda}^\top]$                ▷Initialize the queue with the empty string and its prefix weight vector
  while $Q$ not empty :
    dequeue $\mathbf{b}$ from $Q$
    for $a \in \Sigma$ :                                    ▷Create new sequence by extending with a single symbol
      $\mathbf{b}' \leftarrow \mathbf{b}\mathbf{M}(a)$      ▷Compute new candidate basis vector from sequence
      if $\mathbf{b}' \notin \langle \mathbf{b} \mid \mathbf{b} \in F \rangle$ :   ▷Check if candidate is not in the basis span
        $F \leftarrow F \cup \{\mathbf{b}'\}$               ▷If not, add it to the basis
        append $\mathbf{b}'$ to $Q$
    return $F$
```

checking that it is zero as described by Prop. 3.11.3. The input automata are assumed to be ε-free, so if our input automata have ε-transitions we may need to run ϵ-removal as a preprocessing step, as seen in §3.9.

Algorithm 34 Checking whether two ε-free automata are equivalent.

```
1. def NFAEquivalence($\mathcal{A}_1 = (\Sigma, n_1, \mathbf{M}_1, \boldsymbol{\lambda}_1, \boldsymbol{\rho}_1), \mathcal{A}_2 = (\Sigma, n_2, \mathbf{M}_2, \boldsymbol{\lambda}_2, \boldsymbol{\rho}_2)$):
2.   $\mathcal{A}_- = (\Sigma, n_1 + n_2, \mathbf{M}_-, \boldsymbol{\lambda}_-, \boldsymbol{\rho}_-) \leftarrow$ NFADifference($\mathcal{A}_1, \mathcal{A}_2$)
3.   $F \leftarrow$ ForwardBasisModified($\mathcal{A}_-$)
4.   for $\mathbf{b} \in F$ :
5.     if $\mathbf{b}\boldsymbol{\rho}_- \neq \mathbf{0}$ :
6.       return False
7.   return True
```

Theorem 3.11.18. *Given two ε-free WFSA $\mathcal{A}_1$ and $\mathcal{A}_2$, we can test whether they are equivalent in polynomial time, specifically in $\mathcal{O}(|\Sigma|(n_1 + n_2)^3)$, where n_1 and n_2 are the numbers of states of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively.*

Proof. Firstly, it follows directly from our definition of *equivalence* (Def. 3.11.20) and *difference* (Def. 3.11.22), that two automata are equivalent if and only if their difference is *zero* (Def. 3.11.26). Computing the difference in line 2 of Alg. 34 takes $\mathcal{O}(|\Sigma|(n_1 + n_2)^2)$ as per Prop. 3.11.1. Line 3 then calculates a basis of the forward space in time as shown in prop. It takes linear time to check that the forward space is orthogonal to the final state vector of the difference automaton as shown in Prop. 3.11.3, which we do n times in lines 4-7 with time $\mathcal{O}(n^2)$. Finally, we return False if we find a vector whose inner product with $\boldsymbol{\rho}_-$ is not $\mathbf{0}$, and otherwise return True. Hence, the complexity of the full algorithm is determined by the time it takes to compute the basis of the forward base, and thus equivalence testing takes $\mathcal{O}(|\Sigma|(n_1 + n_2)^3)$. ■

3.11.9 NFA Minimization

In the previous section, we used the first part of Fliess' theorem to show how to test whether two non-deterministic WFSA are equivalent in cubic time. We now turn to the second part of Fliess' theorem, to find the canonical minimum automaton given any ε-free NFA:

Theorem 3.11.19 (Fliess 2)**.** *Let $\mathcal{A}$ be am ε-free WFSA with n states that recognizes the weighted language L with power series r, and let H_r be that language's Hankel matrix. Then there exists an equivalent WFSA $\mathcal{A}_{\text{MIN}}$ that has exactly $n_{\text{MIN}} = rank(H_r)$ states.*

We will prove this part of the theorem by showing how to construct that minimal automaton. Let $\mathcal{A} = (\Sigma, n, \mathbf{M}, \boldsymbol{\lambda}, \boldsymbol{\rho})$ be an ε-free automaton. Let $F \in \mathcal{W}^{\overrightarrow{n} \times n}$ be a matrix whose rows are the basis vectors of the forward space $\mathcal{F}$ of $\mathcal{A}$, where $\overrightarrow{n} \leq n$ as shown in Prop. 3.11.4. Further, let $B \in \mathcal{W}^{n \times \overleftarrow{n}}$ be the matrix with basis vectors of the backward space $\mathcal{B}$ as columns. Since the transition matrix $\mathbf{M}(a)$ is $n \times n$ and F and B are bases of the forward and backward space, respectively, post-/ pre-multiplying with the transition matrix results in row/column vectors that are still in the respective vector spaces:

$$\forall a \in \Sigma : F\mathbf{M}(a) = X \subseteq \mathcal{F}, \text{ and } \mathbf{M}(a)B = Y \subseteq \mathcal{B} \tag{3.292}$$

We can therefore find a mapping $\overrightarrow{\mathbf{M}} : \Sigma \to \mathcal{W}^{\overrightarrow{n} \times \overrightarrow{n}}$ that will map F onto X and another mapping $\overleftarrow{\mathbf{M}} : \Sigma \to \mathcal{W}^{\overleftarrow{n} \times \overleftarrow{n}}$ that maps B onto Y. We can then extend it inductively to words as well:

$$F\mathbf{M}(\boldsymbol{y}) = \overrightarrow{\mathbf{M}}(\boldsymbol{y})F, \text{ and } \mathbf{M}(\boldsymbol{y})B = B\overleftarrow{\mathbf{M}}(\boldsymbol{y}) \tag{3.293}$$

Since F and B have full row/ column rank, the mappings $\overrightarrow{\mathbf{M}}$ and $\overleftarrow{\mathbf{M}}$ are unique. Similarly, we define $\overrightarrow{\boldsymbol{\lambda}} \in \mathcal{W}^{\overrightarrow{n}}$ as the unique vector such that $\boldsymbol{\lambda}^\top F = \boldsymbol{\lambda}$, and $\overleftarrow{\boldsymbol{\rho}} \in \mathcal{W}^{\overleftarrow{n}}$ as the unique vector such that $B\overleftarrow{\boldsymbol{\rho}} = \boldsymbol{\rho}$. Finally, we define $\overrightarrow{\boldsymbol{\rho}} \overset{\text{def}}{=} F\boldsymbol{\rho}$ and $\overleftarrow{\boldsymbol{\lambda}} \overset{\text{def}}{=} \boldsymbol{\lambda}B$.

Definition 3.11.27 (Forward and backward conjugate)**.** *We now call $\overrightarrow{\mathcal{A}} \overset{\text{def}}{=} (\Sigma, \overrightarrow{n}, \overrightarrow{\mathbf{M}}, \overrightarrow{\boldsymbol{\lambda}}, \overrightarrow{\boldsymbol{\rho}})$ the **forward conjugate** of $\mathcal{A}$ with base F, and $\overleftarrow{\mathcal{A}} \overset{\text{def}}{=} (\Sigma, \overleftarrow{n}, \overleftarrow{\mathbf{M}}, \overleftarrow{\boldsymbol{\lambda}}, \overleftarrow{\boldsymbol{\rho}})$ the **backward conjugate** of $\mathcal{A}$ with base B.*

Before we continue, we want to show that the forward and backward conjugate of an automaton are in fact equivalent automata:

Proposition 3.11.5. *Let $\mathcal{A}$ be a WFSA whose power series is r. Then the forward conjugate $\overrightarrow{\mathcal{A}}$ with power series $\overrightarrow{r}$ and the backward conjugate $\overleftarrow{\mathcal{A}}$ with power series $\overleftarrow{r}$ are equivalent, i.e. $r = \overrightarrow{r} = \overleftarrow{r}$.*

Proof. By symmetry, showing the first equality suffices. We have that for all $\boldsymbol{y} = \mathbf{y}_1 \dots \mathbf{y}_k \in \Sigma^*$:

$$\begin{aligned}
\overrightarrow{r}(\boldsymbol{y}) &= \overrightarrow{\boldsymbol{\lambda}}\overrightarrow{\mathbf{M}}(\boldsymbol{y})\overrightarrow{\boldsymbol{\rho}} && && (3.294)\\
&= \overrightarrow{\boldsymbol{\lambda}}\overrightarrow{\mathbf{M}}(\boldsymbol{y})F\boldsymbol{\rho} && \text{(definition of } \overrightarrow{\boldsymbol{\rho}}) && (3.295)\\
&= \overrightarrow{\boldsymbol{\lambda}}F\mathbf{M}(\boldsymbol{y})\boldsymbol{\rho} && \text{(by 3.293)} && (3.296)\\
&= \boldsymbol{\lambda}\mathbf{M}(\boldsymbol{y})\boldsymbol{\rho} && \text{(definition of } \overrightarrow{\boldsymbol{\lambda}}) && (3.297)\\
&= r(\boldsymbol{y}) && && (3.298)
\end{aligned}$$

■

Furthermore, we need to show that we can get to a minimal automaton using the conjugate construction. We first show that the forward and backward conjugate are minimal with respect to their forward and backward spaces, respectively.

Definition 3.11.28 (Forward and backward minimal)**.** *We call an automaton **forward minimal** if it has exactly as many states as its forward space has dimensions. We call an automaton **backward minimal** if it has exactly as many states as its backward space.*

Proposition 3.11.6. *A forward conjugate automaton is* forward minimal*. A backward conjugate automaton is* backward minimal*.*

Proof. Again, we only show the forward case since the proof for the backward conjugate proceeds analogously. Let $\overrightarrow{\mathcal{A}} = (\Sigma, \overrightarrow{n}, \overrightarrow{\mathbf{M}}, \overrightarrow{\boldsymbol{\lambda}}, \overrightarrow{\boldsymbol{\rho}})$ be a forward conjugate of $\mathcal{A}$ with base F. We want to show that $\overrightarrow{\mathcal{A}}$ has no more states than the dimension of its forward space:

$$
\begin{aligned}
&\dim\langle \overrightarrow{\boldsymbol{\lambda}}\overrightarrow{\mathbf{M}}(\boldsymbol{y}) \mid \boldsymbol{y} \in \Sigma^*\rangle && && (3.299)\\
=&\dim\langle \overrightarrow{\boldsymbol{\lambda}}\overrightarrow{\mathbf{M}}(\boldsymbol{y})F \mid \boldsymbol{y} \in \Sigma^*\rangle && \text{(the rows of } F \text{ are independent)} && (3.300)\\
=&\dim\langle \overrightarrow{\boldsymbol{\lambda}}F\mathbf{M}(\boldsymbol{y})F \mid \boldsymbol{y} \in \Sigma^*\rangle && \text{(by 3.293)} && (3.301)\\
=&\dim\langle \boldsymbol{\lambda}\mathbf{M}(\boldsymbol{y}) \mid \boldsymbol{y} \in \Sigma^*\rangle && \text{(definition of } \overrightarrow{\boldsymbol{\lambda}}\text{)} && (3.302)\\
=&\dim \mathcal{F} && \text{(definition of } \mathcal{F}\text{)} && (3.303)\\
=&\overrightarrow{n} && \text{(definition of } \overrightarrow{n}\text{)} && (3.304)
\end{aligned}
$$

■

Proposition 3.11.7. *Let $\mathcal{A}$ be a WFSA. Then a backward conjugate of a forward conjugate $\overrightarrow{\mathcal{A}}$ of $\mathcal{A}$ is minimal, that is, it has as many states as the dimension of its Hankel matrix.*

Proof. Let $\overrightarrow{\mathcal{A}} = (\Sigma, \overrightarrow{n}, \overrightarrow{\mathbf{M}}, \overrightarrow{\boldsymbol{\lambda}}, \overrightarrow{\boldsymbol{\rho}})$ be the forward conjugate of $\mathcal{A}$. Then by Prop. 3.11.6 it is forward minimal, i.e. its forward space has dimension $\overrightarrow{n}$. Let $B \in \mathcal{W}^{\overrightarrow{n} \times \overleftarrow{n}}$ be a matrix whose columns are a basis of the backward space of $\overrightarrow{\mathcal{A}}$. By Thm. 3.11.17, we need to show that $\overleftarrow{n} = rank(H_r)$. Let F_r and B_r be the matrices from the proof of Thm. 3.11.17. Since $\overrightarrow{\mathcal{A}}$ is forward minimal, the columns of F_r are linearly independent. Then:

$$
\begin{aligned}
\overleftarrow{n} &= rank(B) && \text{(definition of } B\text{)} && (3.305)\\
&= rank(F_r B) && (F_r \text{ has lin. indep. columns}) && (3.306)\\
&= \dim\langle F_r \overrightarrow{\mathbf{M}}(\boldsymbol{y})\overrightarrow{\boldsymbol{\rho}} \mid \boldsymbol{y} \in \Sigma^*\rangle && \text{(definition of } B\text{)} && (3.307)\\
&= rank(F_r B_r) && \text{(definition of } B_r\text{)} && (3.308)\\
&= rank(H_r) && \text{(proof of Thm. 3.11.17)} && (3.309)
\end{aligned}
$$

■

Now we just need to find a way to construct the conjugate automata and we have all we need to minimize an NFA! To find the conjugate mappings and vectors and hence the conjugate automata, we will make use of the *right/ left inverse* of matrices:

Definition 3.11.29 (Right and left inverse)**.** *Let $\mathbf{A} \in \mathcal{W}^{m \times n}$ be a matrix. If there exists another matrix $\mathbf{B} \in \mathcal{W}^{n \times m}$ for which $\mathbf{AB} = \mathbf{I}^{m \times m}$, then we call $\mathbf{B}$ a **right inverse** of $\mathbf{A}$, and denote it by $\mathbf{A}_r^{-1}$. Conversely, if there exists a matrix $\mathbf{C} \in \mathcal{W}^{n \times m}$ for which $\mathbf{AC} = \mathbf{I}^{m \times m}$, we call it the **left inverse** of $\mathbf{A}$, denoted by $\mathbf{A}_l^{-1}$.*

Proposition 3.11.8 (Inverse existence)**.** *For any matrix $\mathbf{A} \in \mathcal{W}^{m\times n}$ with full row rank, that is $rank(\mathbf{A}) = m$, there exists a right inverse $\mathbf{A}_r^{-1} \in \mathcal{W}^{n\times m}$. Conversely, for any matrix $\mathbf{B} \in \mathcal{W}^{m'\times n'}$ with full column rank, i.e. $rank(\mathbf{B}) = n'$, there exists a left inverse $\mathbf{B}_l^{-1} \in \mathcal{W}^{n'\times n'}$.*

Proof. We are looking for a matrix $\mathbf{B}\mathcal{W}^{n\times m}$ that satisfies $\mathbf{AB} = \mathbf{I}^{m\times m}$, that is, it solves a system of linear equations with m unknowns. Since we assume $rank(\mathbf{A}) = m$, we have m linearly independent equations, so there must be a unique solution. The second statement follows by symmetry. ■

We can easily verify that the matrix $\mathbf{A}_r^{-1} = \mathbf{A}^\top(\mathbf{A}\mathbf{A}^\top)_r^{-1}$ is such a right inverse (where we use that $\mathbf{A}\mathbf{A}^\top \in \mathcal{W}^{m\times m}$ also has full row rank and therefore has a right inverse):

$$\mathbf{A}\mathbf{A}_r^{-1} = (\mathbf{A}\mathbf{A}^\top)(\mathbf{A}\mathbf{A}^\top)_r^{-1} = \mathbf{I}^{m\times m} \tag{3.310}$$

Note that, since $\mathbf{A}\mathbf{A}^\top$ is a full-rank square matrix, its right inverse exists and is a unique matrix inverse, and can hence be written as $(\mathbf{A}\mathbf{A}^\top)^{-1}$ and computed by Gaussian Elimination[28] in $\mathcal{O}(m^3)$.

Equivalently, for a matrix $\mathbf{B} \in \mathcal{W}^{m\times n}$ with full column rank, i.e. $rank(\mathbf{B}) = n$, a left inverse is given by $\mathbf{B}_l^{-1} = (\mathbf{B}\mathbf{B}^\top)^{-1}\mathbf{B}^\top$, since then:

$$(\mathbf{B}^\top\mathbf{B})^{-1}(\mathbf{B}^\top\mathbf{B}) = \mathbf{I}^{n\times n} \tag{3.311}$$

So with this, we can easily compute the forward conjugate of a WFSA. An algorithm to do this is given in Alg. 35. The algorithm to compute a backward conjugate is very similar and hence omitted here.

Proposition 3.11.9. *Let $\mathcal{A} =$ be a WFSA. A forward conjugate $\overrightarrow{\mathcal{A}}$ of $\mathcal{A}$ and a backward conjugate of $\mathcal{A}$ can be computed in polynomial time, specifically, in $\mathcal{O}(|\Sigma|n^2 + n^3)$.*

Proof. Alg. 35 computes a forward conjugate automaton as described in the parts above. It requires a matrix inversion in line 3, which takes $\mathcal{O}(n^3)$, since $F \in \mathcal{W}^{\overrightarrow{n}\times n}$, and $\overrightarrow{n} = \mathcal{O}(n)$. Apart from this, the most costly part are lines 4-5 where we compute the forward conjugate transition map for each symbol. The matrix multiplication runs in $\mathcal{O}(n^2)$, so the for loop takes $\mathcal{O}(|\Sigma|n^2)$. Hence the runtime of the overall algorithm is as stated. ■

Algorithm 35 Computing a forward conjugate with base F of an automaton.

```
1. def ForwardConjugate($\mathcal{A} = (\Sigma, n, \mathbf{M}, \boldsymbol{\lambda}, \boldsymbol{\rho}), F$):
2.     $\overrightarrow{n} \leftarrow rows(F)$
3.     $F_r^{-1} \leftarrow F^\top(FF^\top)^{-1}$        ▷Compute a right inverse of $F$ by Eq. (3.310)
4.     for $a \in \Sigma$ :
5.         $\overrightarrow{\mathbf{M}}(a) \leftarrow F\mathbf{M}(a)F_r^{-1}$
                                                       ▷Compute conjugate transition map by 3.293
6.     $\overrightarrow{\boldsymbol{\lambda}} \leftarrow \boldsymbol{\lambda}F_l^{-1}$
7.     $\overrightarrow{\boldsymbol{\rho}} \leftarrow F\boldsymbol{\rho}$
8.     return $\overrightarrow{\mathcal{A}} = (\Sigma, \overrightarrow{n}, \overrightarrow{\mathbf{M}}, \overrightarrow{\boldsymbol{\lambda}}, \overrightarrow{\boldsymbol{\rho}})$
```

We can now piece together an algorithm to compute the minimal WFSA. This algorithm is given in Alg. 36.

Proposition 3.11.10. *Given an ε-free WFSA $\mathcal{A}$, Alg. 36 computes a minimal automaton in polynomial time, specifically in $\mathcal{O}(|\Sigma|n^3)$.*

[28] https://en.wikipedia.org/wiki/Gaussian_elimination

Algorithm 36 Computing the minimal automaton.

```
def MinimalNFA($\mathcal{A} = (\Sigma, n, \mathbf{M}, \boldsymbol{\lambda}, \boldsymbol{\rho})$):
    $F \leftarrow$ ForwardBasis($\mathcal{A}$)
    $\overrightarrow{\mathcal{A}} \leftarrow$ ForwardConjugate($\mathcal{A}, F$)
    $B \leftarrow$ BackwardBasis($\overrightarrow{\mathcal{A}}$)
    $\mathcal{A}_{\text{MIN}} \leftarrow$ BackwardConjugate($\mathcal{A}, B$)
    return $\mathcal{A}_{\text{MIN}}$
```

Proof. Correctness follows from Prop. 3.11.5 and Prop. 3.11.7. The time complexity is given by the sum of the complexities of the subroutines, as proven in Prop. 3.11.4 and Prop. 3.11.9. ■

So, we have proven the second part of Fliess' theorem (Thm. 3.11.19), that is, we can not only test for equivalence between two non-deterministic finite-state automata, but also construct the minimal NFA (without determinizing!) in polynomial time, specifically in time cubic in the number of input states. Again, note that the restriction of being ε-free was the only requirement, and we can easily circumvent it by running epsilon removal first as a preprocessing step. This concludes our treatment of equivalence and minimization of finite state automata.

3.12 Regular Expressions

3.12.1 Weighted Regular Expressions

So far in the course, we have mostly dealt with weighted regular languages. We have described them with formal power series and weighted finite-state automata. This section describes the last formalism for describing the weighted regular languages we will consider: **Weighted regular expressions**. In contrast to weighted finite-state machines, weighted regular expressions are a *declarative* formalism for expressing the regular languages—indeed, they closely resemble the notation for formal power series. We discuss the notation used for weighted regular expressions and the constructive proofs of their equivalence to weighted finite-state automata.

Regular Expressions

The main topic of the course so far have been (some classes of) string languages. We spent quite a some time developing a widely used formalism of describing them—finite-state automata. However, a finite-state automaton is not the only way to describe a language. This section introduces another very important formalism—regular expressions. We will define them formally and later show that they if fact, as the name suggests, describe exactly the class of *regular* languages.

We can think of them just as a declarative way of describing *some* language. They are constructed recursively from smaller subexpressions with operations defined on them. Importantly, these operations, which we will define shortly, directly correspond to operations on languages that the expressions represent, which is why we first take a look at those. We already saw one important operation on a language—its Kleene closure. We now define two more:

Definition 3.12.1. *Let L_1, L_2 be languages. We define*

*(i) the **union** of L_1 and L_2, denoted as $L_1 \cup L_2$, as the set of string which are in either of the two languages or in both of them;*

*(ii) the **concatenation** of L_1 and L_2, denoted as $L_1 \circ L_2$ or $L_1 L_2$, as the set $\{\boldsymbol{xy} \mid \boldsymbol{x} \in L, \boldsymbol{y} \in L_2\}$— the set of strings which can be formed by concatenating a string in the first language with a string in the second one.*

As with finite-state automata, we will denote the language represented by a regular expression α with $L(\alpha)$. This allows us to define regular expressions *inductively* based on the languages they define.

Definition 3.12.2 (Regular Expression)**.** *A **regular expression** is defined recursively as:*

- *$\emptyset$, ε, and $a \in \Sigma$ are all regular expressions—they define the languages $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$, $L(a) = \{a\}$;*
- *if α and β are regular expressions, then so are $\alpha + \beta$, $\alpha\beta = \alpha \circ \beta$, α^*, and (α), representing languages $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$, $L(\alpha\beta) = L(\alpha) L(\beta)$, $L(\alpha^*) = L(\alpha)^*$, and $L((\alpha)) = L(\alpha)$.*

The operation $+$ is called the **or** or the union operation. The $\circ$ operation is the concatenation, and the * operation is the Kleene closure of a regular expression. To avoid ambiguity in the languages expressed by regular expressions, we must specify precedence rules on regular expression operations: the Kleene star has the higher precedence, followed by the concatenation operation, and the or operation has the lowest precedence.

Example 3.12.1. *Let $\Sigma = \{a, b\}$. A few examples of regular expressions over the alphabet Σ are a, defining the language $\{a\}$, $a + abab$, defining the language $\{a, abab\}$, and bab^*a, defining the language $\{ba, baba, bababa, \ldots\}$.*

Weighted Regular Expressions

Just like we can extend finite-state automata with weighted finite-state automata, we can also introduce *weighted* regular expressions to be able to declaratively define *weighted* regular languages.

Definition 3.12.3 (Weighted Regular Expression). *Let $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ be a semiring and Σ some alphabet.* ***Weighted regular expressions*** *over $\mathcal{W}$ are defined recursively as:*

- *$\emptyset$, ε, and $a \in \Sigma$ are all regular expressions—they define the same languages as in the unweighted case;*
- *if α and β are regular expressions, then so are $x\alpha$, αx for $x \in \mathbb{K}$, $\alpha + \beta$, $\alpha\beta$, α^*, and (α). Further, $L(x\alpha) = xL(\alpha)$ and $L(\alpha x) = L(\alpha)x$, while the languages of other constructions are the same as in the unweighted case.*

We can see that the only change weighted regular expressions bring to the table are $\mathbb{K}$-valued *weights* associated with (sub)expressions. These represent the weights $\in \mathbb{K}$ associated to the strings represented with the expressions, i.e., the coefficients of the formal power series of the language. As with automata, an unweighted regular expression can be view as a weighted one over the Boolean semiring.

Example 3.12.2. *Let $\Sigma = \{a, b\}$ and $\mathcal{W} = ([0, 1], +, \times, 0, 1)$ the probability semiring. A few examples of weighted regular expressions over the alphabet Σ are $1.0a$, defining the formal power series a, $0.42a + \frac{1}{\pi}abab$, defining the formal power series $0.42a \oplus \frac{1}{\pi}abab$, and $\frac{1}{3}b0.7a\frac{1}{4}b^*0.9a$, defining the language $\left\{0.9 \cdot \frac{1}{3}ba, 0.9 \cdot 0.7 \cdot \frac{1}{3 \cdot 4}baba, 0.9 \cdot 0.7^2 \cdot \frac{1}{3 \cdot 4^2}bababa, \ldots\right\}$.*

Although we mentioned the relation between (weighted) *regular* expression and *regular* languages multiple times above, we have not backed up this connection in any way yet. Indeed, the correspondence between the languages represented with (weighted) regular expressions and those represented with (weighted) finite-state automata is a fundamental result in the theory of formal languages. The next section explores it in more detail.

Thompson's Construction: Converting Regular Expressions to Finite-State Automata

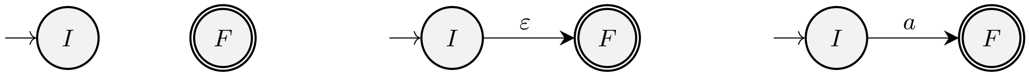
The goal of this section is to show that the language of any (weighted) regular expression can be encoded with a (weighted) finite-states automaton. This will show that regular expressions are at most as expressive as finite-states automata. The next section deals with the opposite transformation. As we have often done in this course, we start with the unweighted case and generalize it to the weighted one.

Unweighted Case. We have to show that, if a language L is encoded by some regular expression α, i.e., $L = L(\alpha)$, then there exists some FSA $\mathcal{A}$ such that $L = L(\mathcal{A})$. The proof will we a constructive one using structural induction on the expression α. That is, we will show how to represent the basic regular expressions as defined in Def. 3.12.2 and then formally describe the construction of automata accepting the languages of regular expressions derived using the 4 operations defined on the expressions. This is formally shown in the following theorem.

Theorem 3.12.1. *Let α be some regular expression and $L = L(\alpha)$. Then $L = L(\mathcal{A})$ for some unweighted automaton $\mathcal{A}$ with exactly one initial state, exactly one final state, no transitions into the initial state and no transitions out of the final state.*

Proof. We proof the theorem using structural induction, relying on the recursive definition of regular expressions.

- **Base case**: The base case constructions are directly shown in Fig. 3.39. Fig. 3.39a shows the automaton accepting only ε—this is easy to see since the only path from a starting state to an accepting one is labeled ε. Similar holds for the automaton in Fig. 3.39b, where the only path is labeled with a. Lastly, we see that the automaton in Fig. 3.39c does not accept any string, meaning that its language is $\emptyset$. Furthermore, it is easy to see that all these automata satisfy properties required by the theorem.



(a) The automaton accepting the language of the regular expression $\alpha = \emptyset$.

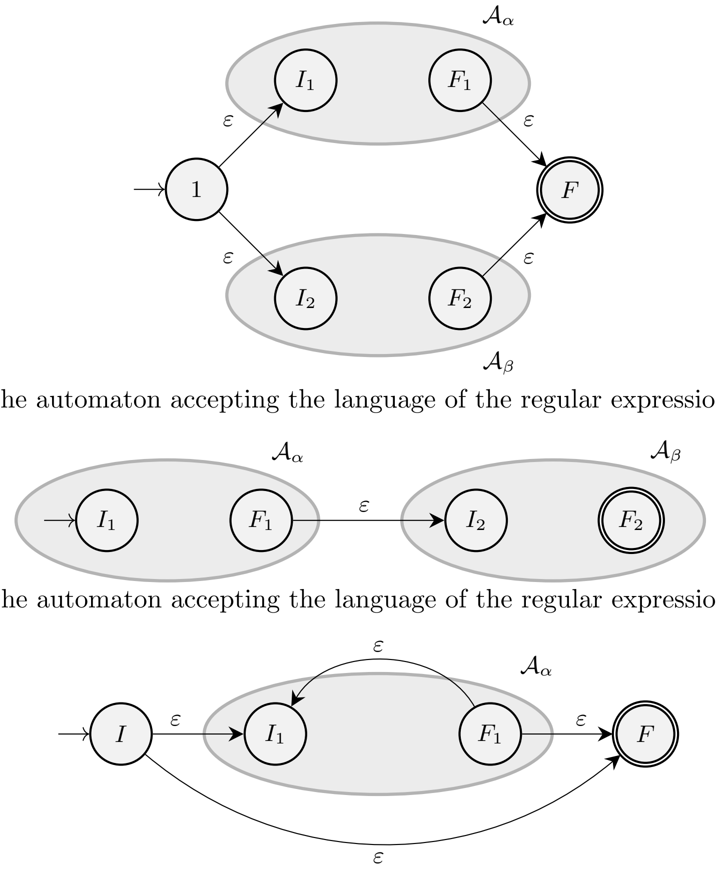
(b) The automaton accepting the language of the regular expression $\alpha = \varepsilon$.

(c) The automaton accepting the language of the regular expression $\alpha = a$ for $a \in \Sigma$.

Figure 3.39: Three base cases for constructing a FSA from a regular expression.

- **Inductive step**: Let α and β be two regular expressions. We now assume we can represent their languages with finite-state automata. These constructions of the automata accepting languages of regular expressions obtained from operations defined on them are shown in Fig. 3.40. Fig. 3.40a shows the automaton of the regular expression $\alpha + \beta$. It is easy to see it accepts exactly the *union* of the languages $L(\alpha)$ and $L(\beta)$. Starting in the new start state, we can ε-transition into a starting state of one of the original automata. Once we arrive to a final state of one of them, we can then ε-transition into the new final state. Thus, $L(\mathcal{A}_1) = L(\alpha) \cup L(\beta)$. Fig. 3.40b shows the automaton corresponding to the regular expression $\alpha\beta$. Observe that no new state is added and the starting state of the first automaton becomes the starting state of the combined one while the final state of the second one becomes the final state in the constructed one. Again, it is easy to see that a string $\boldsymbol{y}$ will be accepted by the automaton shown in Fig. 3.40b if and only if there is a substring in $\boldsymbol{y}$ which leads from the initial to the final state of the first automaton and the rest of the string leads from the second initial state to the second final one. This means that the constructed automaton accepts the language $L(\alpha)L(\beta)$. The construction of the automaton corresponding to α^* is only slightly more complicated. It is shown in Fig. 3.40c. The automaton allows us to either jump from the initial state to the final one directly (meaning that its language contains ε) or to move from its initial state to the initial state of the original automaton, move through it, and then loop back to the initial state from the final one. This means that its language contains the strings obtained by an arbitrary number of concatenations of the string in the language of the original automaton. Together, this means that its language is exactly $(L((\alpha))^*$. Lastly, the construction of the automaton for the expression (α) is trivial, since the parentheses do not change the language. Thus, the original automaton can be returned.

■



(a) The automaton accepting the language of the regular expression $\alpha + \beta$.

(b) The automaton accepting the language of the regular expression $\alpha\beta$.

(c) The automaton accepting the language of the regular expression α^*.

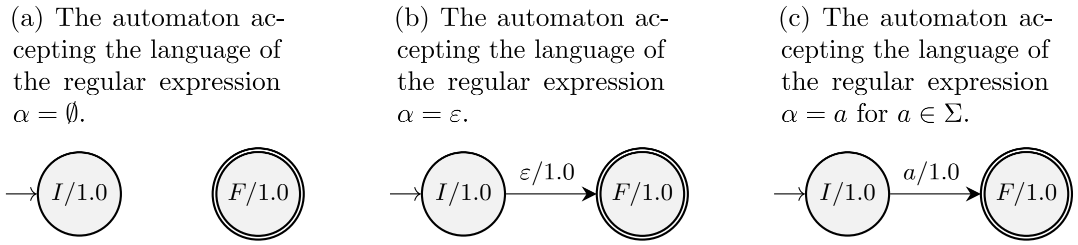
Figure 3.40: Three inductive cases for constructing a FSA from a regular expression. $\mathcal{A}_\alpha$ denotes the WFSA corresponding to the regular expression α.

This recursive construction of automata corresponding to a regular expression is known as **Thompson's construction**.

Weighted Case. We now discuss the generalization of the unweighted Thompson's construction to the weighted case. The entire construction will be almost exactly the same, we will only have to put the weights in the appropriate places. Similarly as before, the following theorem formalizes the construction.

Theorem 3.12.2. *Let α be some weighted regular expression and $L = L(\alpha)$ over the semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$. Then $L = L(\mathcal{A})$ for some weighted automaton $\mathcal{A}$ with exactly one initial state, exactly one final state, no transitions into the initial state and no transitions out of the final state.*

Proof. The proof follows the unweighted construction closely. We only have to properly assign weights to the transitions. **Base case**: The construction in the base case, $(\emptyset, \varepsilon, a \in \Sigma)$ is exactly the same as in the unweighted construction, except that we additionally *weight* all the transitions as well as the initial and final states with the semiring weight $\mathbf{1}$. The constructions are shown in Fig. 3.41. It is easy to see that these automata satisfy the conditions on the number of final and initial states and that the base strings are then accepted with the weight of $\mathbf{1}$. **Inductive step**: Let now α and β be two weighted regular expressions. Again, the construction of the cases covered

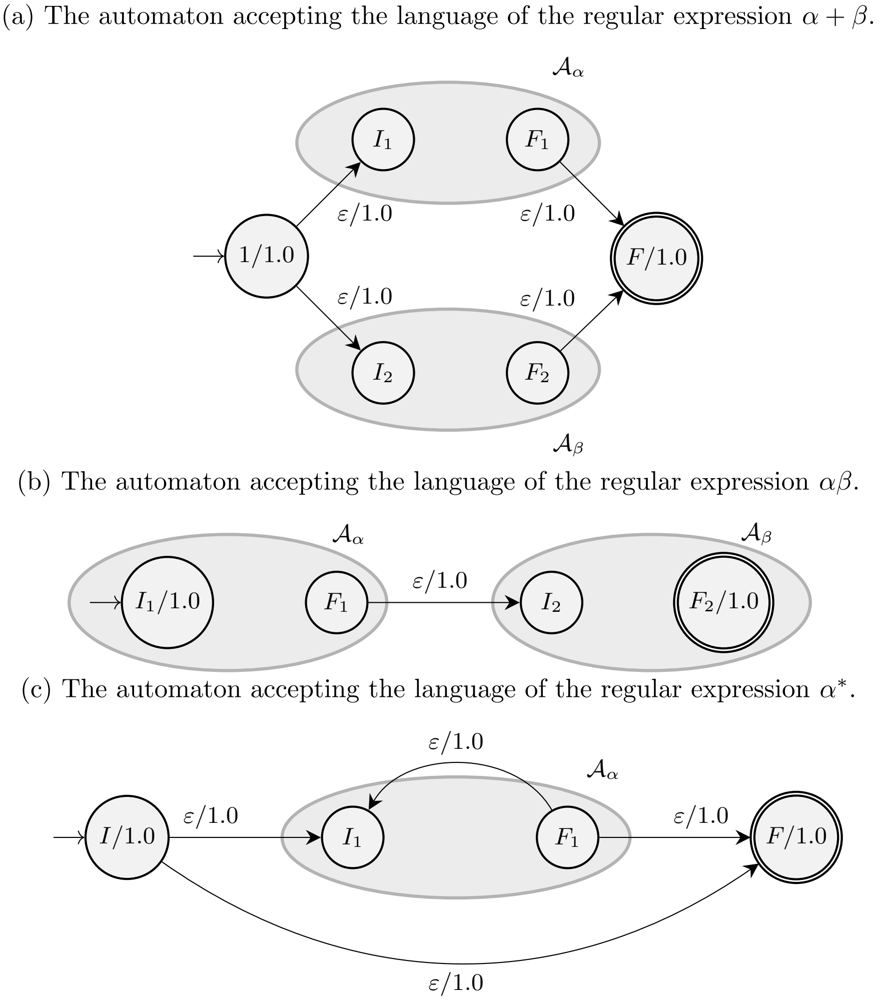

(a) The automaton accepting the language of the regular expression $\alpha = \emptyset$.

(b) The automaton accepting the language of the regular expression $\alpha = \varepsilon$.

(c) The automaton accepting the language of the regular expression $\alpha = a$ for $a \in \Sigma$.

Figure 3.41: Three base cases for constructing a WFSA from a weighted regular expression.

in Thm. 3.12.1 (concatenation, union and Kleene star, and the expression in parentheses) is almost identical, as shown inFig. 3.42. The only difference is the weighting of the new new transitions, and the new initial and final states with the semiring **1** in the case of addition and Kleene closure. In the case of concatenation, we set the final weight of the final state of the first automaton to **0**, and we do the same with the initial weight of the initial state of the second automaton. Again, the new transitions are weighted with **1**. We do not add any new states in that case. The only additional


(a) The automaton accepting the language of the regular expression $\alpha + \beta$.

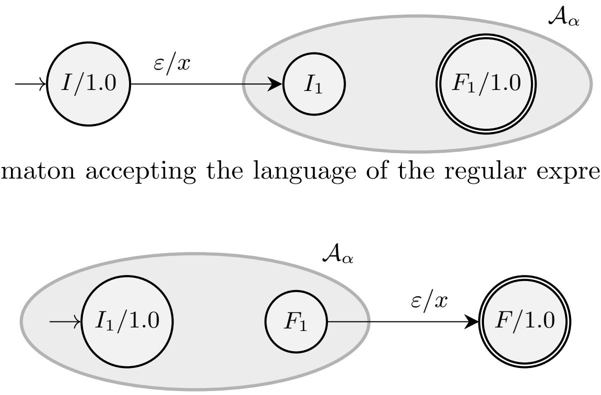
(b) The automaton accepting the language of the regular expression $\alpha\beta$.

(c) The automaton accepting the language of the regular expression α^*.

Figure 3.42: Three inductive cases for constructing a WFSA from a weighted regular expression. $\mathcal{A}_\alpha$ denotes the WFSA corresponding to the regular expression α.

construction we have to consider are the constructions of $x\alpha$ and αx for $x \in \mathbb{K}$. Let $\mathcal{A}$ be the automaton corresponding to the regular expression α. By out inductive hypothesis, is has exactly one initial and one final state. We can construct the automaton corresponding to $x\alpha$ by introducing a new initial state and connecting it via a x-weighted transition with the initial state of the smaller automaton, as shown in Fig. 3.43a. This has the effect of pre-$\otimes$-multiplying all strings accepted by $\mathcal{A}$ with x, meaning that the language accepted by the constructed automaton is exactly $xL(\alpha)$. Similarly, to obtain the automaton corresponding to αx, we can introduce a new *final* state and connecting the original final state with it via a x-weighted transition. Constructing multiplications this way avoids changing the initial and final weights, which allows for easy combination of two automata when taking the concatenation and union. We result in an automaton accepting identical strings as $\mathcal{A}$, with the weights of post-$\otimes$-multiplied with x, meaning that its language is $L(\alpha)x$. This concludes the proof. ■

(a) The automaton accepting the language of the regular expression $x\alpha$ for $x \in \mathbb{K}$.



(b) The automaton accepting the language of the regular expression αx for $x \in \mathbb{K}$.

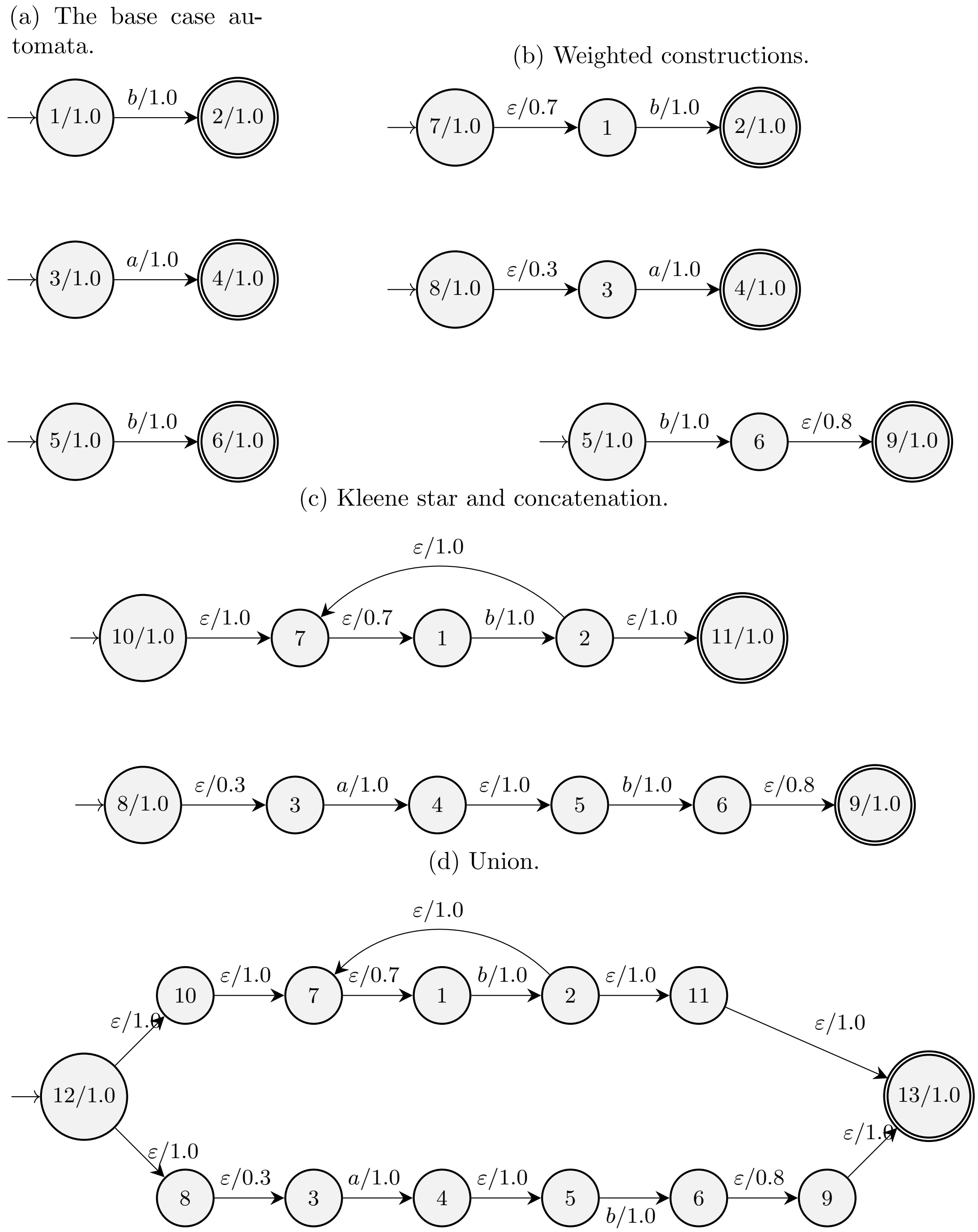
Figure 3.43: The two cases of constructing a WFSA from a weighted regular expression when multiplying a regular expression with semiring weight. $\mathcal{A}_\alpha$ denotes the WFSA corresponding to the regular expression α.

Example 3.12.3. *Let us show this construction through a concrete example. Let* $\alpha = (0.7b)^* + 0.3ab0.8$ *be a regular expression over the real semiring. We will construct the corresponding WFSA by following the inductive construction steps outlined in the proof of Thm. 3.12.2. We start by constructing the base automata: the symbol* a *and the two symbols* b*. These are shown in Fig. 3.44a. Next, we add the weights to the base automata, as shown in Fig. 3.44b. These are then concatenated and asterated as shown in Fig. 3.44c and finnaly added together as depicted in Fig. 3.44d.*

The theorems in this section not only prove one direction of the correspondence of (weighted) automata and (weighted) regular expressions. Importantly, they are also constructive, meaning that they explicitly propose how we can obtain these automata. Luckily, the proof in the reverse direction will be constructive as well, albeit a bit more involved.

Kleene's Algorithm: Converting Finite-State Automata to Regular Expressions

The previous section showed that regular expressions are at most as powerful as finite-state automata. This section shows that they are indeed equally as powerful by showing that any language described by a (weighted) finite-state machine can also be represented by a (weighted) regular expression.


(a) The base case automata.

(b) Weighted constructions.

(c) Kleene star and concatenation.

(d) Union.

Figure 3.44: The process of creating the WFSA corresponding to the regular expression $(0.7b)^* + 0.3ab0.8$.

This direction is a bit more involved than its opposite. However, we will see that, for the most part, we will again be able to reuse a lot of the machinery we already presented in the course, with some specializations to the specific problem at hand. This time, we will discuss both the unweighted and the weighted case together. That is because the construction of the regular expression will be done with the same algorithm in both cases, with the unweighted case reducing to the Boolean semiring, as you might imagine.

The idea of the construction is to build up the regular expression from sub-expressions corresponding to subparts of the input automaton. Intuitively, since the labels of the transitions along a path are *concatenated* to for a string, we will concatenate them in the regular expression as well. Furthermore, since the alternative paths lead to different strings in the language of the automaton, we will combine them using the + operation in the resulting regular expression. Looking at the task that way, it begins to look very similar to another very general problem we already discusses—the pathsum. We will obtain the regular expressions by concatenating ("$\otimes$-multiplying) labels of the transitions along the paths and taking the union of different possible strings along all possible paths ($\oplus$-summation). This suggests we can very cleanly encode this as a *pathsum problem* on a specific semiring and an automaton we can trivially obtain from the input one. These ideas will be used in the proof of the main theorem below. For that, we will need a new semiring, one in which the operations we perform will correspond somehow to the operations on regular expressions. This is the so-called **Kleene semiring**. As we require this construction to work on arbitrary automata, including the cyclic ones, the pathsum has to be taken over a *closed* semiring. Indeed, as we will see below, the Kleene semiring has a very natural, and familiar, Kleene closure operator.

Definition 3.12.4 (Kleene Semiring). *The **Kleene semiring**, also called the **regular expression** semiring, $\mathcal{K} = \left(\Sigma^*, \oplus^{(\mathcal{K})}, \otimes^{(\mathcal{K})}, \mathbf{0}^{(\mathcal{K})}, \mathbf{1}^{(\mathcal{K})}, {}^{*(\mathcal{K})}\right)$ over an alphabet Σ, is defined as:*

- $\oplus^{(\mathcal{K})} = +$ *(regular expression or);*
- $\otimes^{(\mathcal{K})} = \circ$ *(concatenation);*
- ${}^{*(\mathcal{K})} = {}^*$ *(the regular expression Kleene star);*
- $\mathbf{0}^{(\mathcal{K})} = \emptyset$*;*
- $\mathbf{1}^{(\mathcal{K})} = \varepsilon$*.*

With the semiring defined, finding the regular expression corresponding to an input automaton is just a matter of pre-processing the input automaton slightly and running the pathsum on the modified machine. To prepare the automaton, we *encode* it over a *new* semiring, which is obtained by "merging" the weights of the original automaton and the transition labels. Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be an automaton over some semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$. Then, we construct the modified automaton $\mathcal{A}_{\text{REGEX}} = (\Sigma, Q, I, F, \delta_{\text{REGEX}}, \lambda_{\text{REGEX}}, \rho_{\text{REGEX}})$ over the a new semiring $\mathcal{W}_{\text{REGEX}}$, where the weights of the transitions are constructed such that

$$q \xrightarrow{a/w} q' \in \delta \iff q \xrightarrow{a/wa} q' \in \delta_{\text{REGEX}}. \tag{3.312}$$

Additionally, we modify the initial weights w_I and the final weights w_F into $w_I\varepsilon$ and $w_F\varepsilon$, respectively.

With the automaton constructed, we can state and prove the following result:

Definition 3.12.5. *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA. Let $\mathcal{A}_{\text{REGEX}}$ be the augmented automaton constructed by merging the input symbols and weights as described above. Then $\alpha = Z(\mathcal{A}_{\text{REGEX}})$ is a regular expression describing the same language as $\mathcal{A}$.*

Proof. Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA and $\mathcal{A}_{\text{REGEX}}$ the preprocessed version of $\mathcal{A}$ as described above. Let $\alpha \overset{\text{def}}{=} Z(\mathcal{A}_{\text{REGEX}}) = \bigoplus^{(\mathcal{K})}_{\boldsymbol{\pi} \in \Pi(\mathcal{A}_{\text{REGEX}})} \lambda_{\text{REGEX}}\left(p\left(\boldsymbol{\pi}\right)\right) \otimes w_{\text{REGEX}}\left(\boldsymbol{\pi}\right) \otimes \rho_{\text{REGEX}}\left(n\left(\boldsymbol{\pi}\right)\right)$ be the pathsum of the preprocessed automaton. By the definition of the the $\oplus$ and $\otimes$ operations, this is clearly a valid regular expression. We have to show that $L(\alpha) = L(\mathcal{A})$. Note that the language of the original automaton, $L(\mathcal{A})$ is by definition the union (formal power series addition) of the formal power series realized by the all the paths in the automaton (pre- and post- multiplied with the initial and final weights, respectively). Consider now a path $\boldsymbol{\pi}_{\text{REGEX}}$ in $\mathcal{A}_{\text{REGEX}}$ and its corresponding path $\boldsymbol{\pi} = q_1 \xrightarrow{a_1/w_1} q_2, q_2 \xrightarrow{a_2/w_2} q_3, \dots, q_{n-1} \xrightarrow{a_{n-1}/w_{n-1}} q_n$ in $\mathcal{A}$. $\boldsymbol{\pi}$ realizes the the formal power series $w_1 \otimes \dots \otimes w_{n-1} a_1 \dots a_{n-1}$. By the definition of the weights in $\mathcal{A}_{\text{REGEX}}$, we then have

$$w_{\text{REGEX}}(\boldsymbol{\pi}) = \bigotimes^{(\mathcal{K})}_{q \xrightarrow{a/w} q' \in \mathcal{E}_{\mathcal{A}}(q)} aw, \tag{3.313}$$

i.e., its weight is the regular expression $w_1 a_1 w_2 a_2 \dots w_{n-1} a_{n-1}$ representing the formal power series $w_1 \otimes \dots \otimes w_{n-1} a_1 \dots a_{n-1}$. Pre-$\otimes^{(\mathcal{K})}$-multiplying the $\mathcal{A}_{\text{REGEX}}$-weight of $\boldsymbol{\pi}_{\text{REGEX}}$ with the initial weight and post-$\otimes^{(\mathcal{K})}$-multiplying it with the defined final weight results in the weight and regular expression $w_I \varepsilon w_1 a_1 w_2 a_2 \dots w_{n-1} a_{n-1} w_F \varepsilon = w_I w_1 a_1 w_2 a_2 \dots w_{n-1} a_{n-1} w_F$. This represents the formal power series $w_I \otimes w_1 \otimes \dots \otimes w_{n-1} \otimes w_F a_1 \dots a_{n-1}$, exactly the same formal power series as the one represented by the original path in $\mathcal{A}$ pre- and post-$\otimes$ multiplied with the initial and final weights, respectively. This shows that all path weights in the augmented automaton result in a regular expression representing the same language as the path in the original automaton. Since the $\oplus^{(\mathcal{K})}$ represents the *union* of regular expressions, or the *addition* of their formal power series, we conclude that the pathsums over all paths in the automaton $\mathcal{A}_{\text{REGEX}}$ results in the regular expression representing $L(\mathcal{A})$. ■

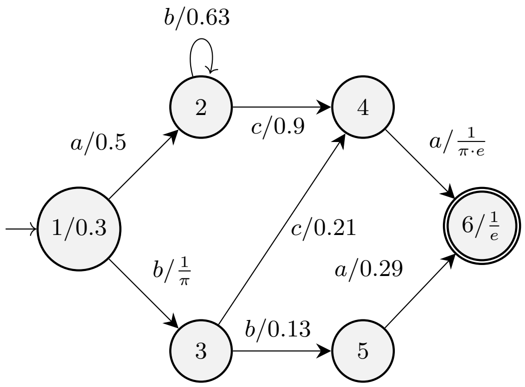
This shows that weighted finite-state automata are at most as powerful as weighted regular expressions. Therefore, this result, together with the previous section, establishes the fundamental equivalence between (weighted) finite-state automata and (weighted) regular languages. As always, the case of unweighted automata reduces to the case where the semiring is Boolean. The described algorithm for constructing a (weighted) regular expression from a (weighted) finite-state automaton is called **Kleene's algorithm**, but as we see, it is just an instance of the pathsum. Note that the exact regular expression constructed by pathsum depends on the pathsum algorithm used. The language described by the regular expressions will, however, be the identical.

Example 3.12.4. *Let us see the conversion of a WFSA to a WRE in action. Consider the WFSA in Fig. 3.45a from* **??** *over the real semiring. We will build the regular expression α representing the same language as $\mathcal{A}$. The augmented automaton $\mathcal{A}'$ is shown Fig. 3.45b. We see that the weights now contain both the original real weight and the input symbol on the transition. There are two accepting paths in $\mathcal{A}'$* without *cycles: the path going through* $1, 2, 4, 6$ *and the one going through* $1, 3, 5, 6$. *We can therefore add the regular expressions corresponding to these two paths straight to α. We can construct them by simply concatenating the labels and the initial/final weight along the path to get $0.3\frac{1}{\pi}b0.21c\frac{1}{\pi \cdot e}a\frac{1}{e} + 0.3\frac{1}{\pi}b0.13b0.29a\frac{1}{e}$. To take care of the cycle at node* 2*, we concatenate the Kleene closure of its weight with the regular expression of the path up to* 2 *pre-concatenated with the initial weight and the path from* 2 *to* 6 *post-concatenated with the final weight, resulting in the regular expression $0.3 \cdot 0.5a0.63b^{*}0.9c\frac{1}{\pi \cdot e}a\frac{1}{e}$. These three options cover all accepting paths in $\mathcal{A}'$. Additively combining these three regular expressions, we get the the final expression*

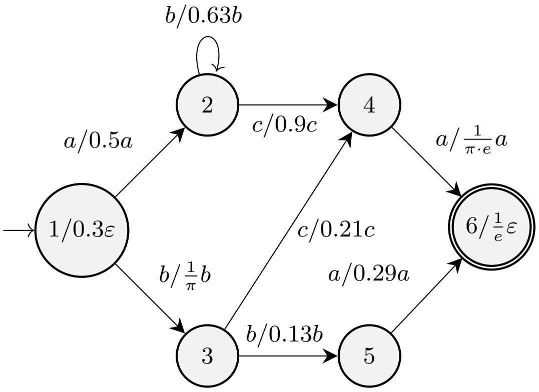
$$\alpha = 0.3\frac{1}{\pi}b0.21c\frac{1}{\pi \cdot e}a\frac{1}{e} + 0.3\frac{1}{\pi}b0.13b0.29a\frac{1}{e} + 0.3 \cdot 0.5a0.63b^{*}0.9c\frac{1}{\pi \cdot e}a\frac{1}{e}. \tag{3.314}$$

We conclude this section by pointing out that the construction just described may be very inefficient and can produce very long (exponentially long) regular expressions. There exists an alternative construction of regular expressions based on *eliminating states* of the input automaton (Hopcroft et al., 2006). It results in shorter expressions, but we do not cover it in the course.

(a) Original automaton $\mathcal{A}$.



(b) Augmented automaton $\mathcal{A}'$ with modified weights.



3.13 (Generalized) Dijkstra's Algorithm

3.13.1 Superior Semirings and Generalized Dijkstra's Algorithm

This section is motivated by the observation that for computing the pathsum of an automaton (the sum of the weights of all accepting paths), we do not need to calculate the all-pairs shortest distance matrix as we have always done so far. It is enough to compute the single-source distances from the initial states of the machine! This is attractive since this is a much easier problem to solve and there exist faster algorithms for computing the distances. One of the best-known and most general ones is the BellmanFord algorithm which correctly computes the single-source shortest paths on any graph without negative cycles. However, its quadratic runtime in the number of states may still be prohibitively slow. Viterbi (forward) algorithm is an order of magnitude faster, but it only works on *acyclic* graphs due to its reliance on processing nodes in topological order. Dijkstra's algorithm strikes the balance between the two algorithms. It is a faster algorithm than BellmanFord, and it also works in the cyclic case. However, it processes nodes in a *best-first* fashion, meaning that it only works on graphs with nonnegative edge weights. We will see that this requirement can easily be expressed as a certain class of semirings! This section discusses a generalization of Dijkstra'a algorithm for computing single-source shortest paths. As was the case for all-pairs shortest paths, the min operation in Dijkstra's algorithm can just be seen as a $\oplus$ operation in the Tropical semiring. However, due to the nature of how Dijkstra's selects nodes to relax[29], it can not be generalized to just any semiring—we will need some notion of order in the semiring and some additional properties on how the semiring multiplication affects the order. We start by introducing a new class of useful semirings, called **superior** semirings, and then discuss Dijkstra's algorithm and its generalization to superior semirings. Again, since Dijkstra is a general weighted graph algorithm, we leave out the interpretation of the graph as an automaton an focus on just the graph concepts. V will denote the set of vertices of the graph and E the set of edges.

?? introduced the notion of *dioids* as the semirings in which the canonical order is in fact a valid order relation. We now look at it more closely in the case of *idempotent* semirings. It turns out that there is a useful alternative formulation of the canonical ordering for this case of semirings. We will make use of it when we talk about generalizing Dijkstra.

[29] **Relaxation** refers to the selection and processing of a node in a graph. In the case of Dijkstra, when relaxing a node q, we mark the optimal path to q *and* update the priorities of q's children.

Lemma 3.13.1. *In an idempotent semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$, the canonical order relation as defined in Def. 3.6.16 can equivalently be characterized by*

$$a \preccurlyeq b \iff a \oplus b = b \tag{3.315}$$

Proof. By definition, we have $a \preccurlyeq b \iff \exists c: \; a \oplus c = b$. Therefore

$$a \oplus b = a \oplus a \oplus c = a \oplus c = b. \tag{3.316}$$

■

Example 3.13.1. *To help the intuition, behind the alternative characterization, consider the Tropical semiring, which is idempotent and superior. Lemma 3.13.1 states that $a \preccurlyeq b$ if and only if $a \oplus b = b$, which in the Tropical semiring means $\min(a, b) = b$. Notice that this means $b \leq a$ in the field of the reals! The canonical order relation $\preccurlyeq$ therefore orders the elements in the opposite direction to what we may expect. Therefore, it is better to think of the relation $a \preccurlyeq b$ as meaning that b is in some sense "better" than a.*

Example 3.13.2. *The intuition behind the relation $\preccurlyeq$ as originally defined in Def. 3.6.16 is easier in the (non-idempotent) semiring of natural numbers $(\mathbb{N}, +, \times, 0, 1)$. There, we have $a \preccurlyeq b \iff \exists c \in \mathbb{N}: \; a + c = b$, which is obviously true only if $a \leq b$ in our "normal" understanding of "smaller".*

The relation $\preccurlyeq$ is in a general dioid still only a *partial* order, which means that not all elements of the set in question may be comparable. Whenever we want to compare arbitrary elements of $\mathbb{K}$, we need to operate with *total* orders. If the natural order of a dioid $\mathcal{W}$ is total, we say that $\mathcal{W}$ is **totally-ordered**. [30] Since we can compare arbitrary elements in a totally-ordered set, we can talk about the minimal elements and, most importantly for us, some sorts of "minimal" paths. [31] The superiority property of a semiring is the abstract form of the *non-negative weights* requirement in shortest-path problems Huang (2008), allowing the use of best-first order algorithms such as Dijkstra's.

Definition 3.13.1 (Superior semiring)**.** *Let $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ be a semiring and $\preccurlyeq$ a partial order over $\mathbb{K}$. We say that $\mathcal{W}$ is* ***superior*** *if $\forall x, y \in \mathbb{K}$:*

$$x \otimes y \preccurlyeq x \tag{3.317}$$

$$x \otimes y \preccurlyeq y. \tag{3.318}$$

In words, this means that the combination of two elements always gets *worse* in the intuition from the previous example. For instance, when searching for the shortest path in a graph with non-negative weights, each edge you visit *increases* the weight of the path, making it worse.

Example 3.13.3. *The tropical semiring $(\mathbb{R}_+, \min, +, \infty, 0)$, the probability semiring $([0, 1], +, \times, 0, 1)$, and the Boolean semiring $(\{0, 1\}, \vee, \wedge, \text{TRUE}, \text{FALSE})$ are all superior. The semiring of the positive reals $(\mathbb{R}_+, +, \times, 0, 1)$ (which is a dioid in contrast to the semiring of all reals $(\mathbb{R}, +, \times, 0, 1)$), is not superior, since the $\otimes$-products can increase.*

Another important semiring property for this discussion is **monotonicity**.

[30] To be more consistent with the rest of the notes at the cost of being less precise, this section will refer to dioids simply as semirings. The fact that the natural order relation is an order relation is assumed throughout the section.

[31] Note that this is different from the minimality in the Tropical semiring. There, the minimum is just the $\oplus$-operation in the semiring. Here, however, we are talking about the minimum of the abstract canonical relation on semirings.

Definition 3.13.2 (Monotonicity)**.** *A semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ with the canonical order relation $\preccurlyeq$ is* ***monotonic*** *if $\forall x, y, z \in \mathbb{K}$:*

$$x \preccurlyeq y \implies x \otimes z \preccurlyeq y \otimes z \tag{3.319}$$

$$x \preccurlyeq y \implies z \otimes x \preccurlyeq z \otimes y. \tag{3.320}$$

Monotonicity is an important property when constructing solutions to optimization problems using dynamic programming since it allows us to reason about the order of composite solutions.

Idempotent semirings are always monotonic, as the following lemma shows.

Lemma 3.13.2. *Let $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ be an* idempotent *semiring. Then $\mathcal{W}$ is monotonic.*

Proof. Suppose $x \preccurlyeq y$, i.e., $x \oplus y = y$. Then, we have:

$$y \otimes z = (x \oplus y) \otimes z = x \otimes z \oplus y \otimes z. \tag{3.321}$$

By the characterization of $\preccurlyeq$ in idempotent semirings, this means that $x \otimes z \preccurlyeq y \otimes z$. The proof for the multiplication from the left is identical. ■

We now state a simple lemma that will help us with the interpretation of the algorithms we consider in this section.

Lemma 3.13.3 (Negative Boundedness)**.** *Let $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ be a superior semiring and $\preccurlyeq$ the canonical order over $\mathbb{K}$. Then for any $x \in \mathbb{K}$:*

$$\boldsymbol{0} \preccurlyeq x \preccurlyeq \boldsymbol{1} \tag{3.322}$$

Proof. For any $x \in \mathbb{K}$, we have $x = \mathbf{1} \otimes x \preccurlyeq \mathbf{1}$ by superiority (and the fact that $\mathbf{1}$ is the identity). On the other hand, we also have $\mathbf{0} = \mathbf{0} \otimes x \preccurlyeq x$ by superiority as well (and the fact that $\mathbf{0}$ is an annihilator). ■

The property is called **negative boundedness**. It intuitively expresses the "direction" of optimization from $\mathbf{0}$, the initial value, to $\mathbf{1}$, the optimal one.

Example 3.13.4. *In the case of the Tropical semiring, when computing the shortest path, we initialize all distances to ∞ ($\boldsymbol{0}$ in the tropical semiring) and optimize them (make them shorter) towards* 0 *($\boldsymbol{1}$ in the tropical semiring).*

Generalized Dijkstra's Algorithm

Although it is not always presented this way, Dijkstra's algorithm is also an instance of dynamic programming. In its basic form, it is used to find the *shortest* path from a *source* node to any other node in the graph. However, as we did when computing all-pairs shortest paths in **??**, the notion of "shortest" can be generalized to any $\oplus$-sum in a superior semiring—the only thing we need to change are the min and + operations for the general semiring $\oplus$ and $\otimes$. Alg. 37 presents the generalized version of Dijkstra's algorithm. The complexity of the algorithm is obviously the same as the original version—$\mathcal{O}(|V| \log |V|)$ when using a Fibonacci heap and $\mathcal{O}((|V| + |E|) \log |V|)$ when using the binary heap. It is well known that Dijkstra's algorithm works correctly on any graphs with nonnegative real weights. In fact, all we need is the superiority and idempotency properties of a semiring, as we show next.

We will prove the correctness of Dijkstra in several steps. First, we have the following lemma.

Algorithm 37 The Dijkstra algorithm.

```
1.  def Dijkstra($\mathcal{G}, s$):
2.    ▷ Initialization
3.    for $q \in \mathcal{G}$ :
4.      $\Delta(q) \leftarrow \mathbf{0}$
5.    $\Delta(s) \leftarrow \mathbf{1}$
6.    queue ← vertices $(\mathcal{G})$                ▷The priority queue prioritized by the values of $\Delta$
7.    while |queue| $> 0$ :
8.      pop the maximal (best) element $q$ off queue
9.      for $q \xrightarrow{w} q' \in \mathcal{E}(q)$ :
10.       $\Delta(q') \leftarrow \Delta(q') \oplus \Delta(q) \otimes w\left(q \xrightarrow{w} q''\right)$
11.       increase priority of $q'$ in queue
12.   return $\Delta$
```

Lemma 3.13.4. *Let $\mathcal{G}$ be some weighted graph over a superior semiring and let q, q' be nodes in $\mathcal{G}$. Then, if the node q is taken off* queue *on Line* 8 *in Alg.* 37 before q', *it holds that* $\Delta(q') \preccurlyeq \Delta(q)$ at any later *point of the algorithm.*

Proof. At the initialization, both q and q' are together in queue. Whichever is taken off first must have had at that point the higher of the two Δ-values. We just have to show that $\Delta(q')$ does not improve beyond $\Delta(q)$ at later iterations of the algorithm, which we can do by induction on the number of nodes processed *after* q has been taken off queue. **Base case**: Let q'' be the node taken off queue the first iteration after q. Then, we know that since q was taken off queue at the previous iteration instead of q'', we had $\Delta(q'') \preccurlyeq \Delta(q)$. During the processing of q on Lines 9–11, $\Delta(q'')$ may have stayed the same (case 1) or it may have been updated to $\Delta(q'') \oplus \Delta(q) \otimes w\left(q \xrightarrow{w} q''\right)$ (case 2). In both cases, it still holds for the updated $\Delta(q'')$ that $\Delta(q'') \preccurlyeq \Delta(q)$. In case 1, this is trivial to see. In case 2, the result is a consequence of the superiority of the semiring.[32] **Inductive step**: Suppose that our hypothesis holds for the first $n-1$ nodes taken off queue. Let q'' be the n-th node taken off queue after q. Similarly as in the base case, $\Delta(q'')$ may be updated a number of times when processing the nodes taken off queue *before* q''. By the inductive hypothesis, all the previously removed nodes have their Δ-values $\preccurlyeq$-lower than q. We can follow identical reasoning as in the base case to show that $\Delta(q'')$ does not improve beyond $\Delta(q)$. In particular, the above holds for $q'' = q'$. ■

This allows us to state the following simple corollary.

Lemma 3.13.5. *Let $\mathcal{G}$ be some weighted graph over a superior idempotent semiring and let q be a node in $\mathcal{G}$. Then, after q is taken off* queue *on Line* 8 *in Alg.* 37 *in some iteration of the while loop, its value of Δ, $\Delta(q)$, does not change anymore.*

Proof. This is a consequence of Lemma 3.13.4. Let q' be a node taken off queue after q. By Lemma 3.13.4 we know that $\Delta(q') \preccurlyeq \Delta(q)$, or equivalently $\Delta(q') \oplus \Delta(q) = \Delta(q)$. The value of $\Delta(q)$ gets updated on Line 10 as $\Delta(q) \leftarrow \Delta(q) \oplus \Delta(q') \otimes w\left(q' \xrightarrow{w} q\right)$ if there is an edge from q' to q. However, due to superiority and idempotency, we have

$$\Delta(q) \leftarrow \Delta(q) \oplus \Delta(q') \otimes w\left(q' \xrightarrow{w} q\right) \preccurlyeq \Delta(q) \oplus \Delta(q') = \Delta(q). \tag{3.323}$$

[32] More precisely, we also need $a \preccurlyeq b, a \preccurlyeq c \implies a \preccurlyeq b \oplus c$, which is trivial to show.

This means that $\Delta(q)$ does not change anymore. ■

Lemma 3.13.6. *Let $\mathcal{G}$ be a weighted graph over a superior idempotent semiring. At any iteration of* Dijkstra*, we have*

$$\Delta\left(q\right) \preccurlyeq Z\left(s, q\right), \tag{3.324}$$

i.e., the current estimate of the pathsum is at most as good as the actual pathsum.

Proof. Again, with induction on the number of nodes processed, n.

- **Base case**: $n = 0$. Since all Δ-values apart from that of s (whose Δ-value is one, which equals the pathsum) are initialized to $\mathbf{0}$, the base case holds by negative boundedness.
- **Inductive step**: Suppose the hypothesis holds for the first $n-1$ processed nodes. Let q be the n^{th} processed node. Then, its Δ-value will be $\Delta(q) = \bigoplus_{q' \in V \setminus \mathsf{queue}} \Delta(q') \otimes w\left(q' \xrightarrow{w} q\right)$. By the inductive hypothesis, $\Delta(q') \preccurlyeq Z(s, q')\ \forall q' \in V \setminus \mathsf{queue}$. Therefore,

$$\Delta(q) = \bigoplus_{q' \in V \setminus \mathsf{queue}} \Delta(q') \otimes w\left(q' \xrightarrow{w} q\right) \tag{3.325}$$

$$\preccurlyeq \bigoplus_{q' \in V \setminus \mathsf{queue}} Z(s, q') \otimes w\left(q' \xrightarrow{w} q\right) \tag{3.326}$$

$$\preccurlyeq \bigoplus_{q' \in V} Z(s, q') \otimes w\left(q' \xrightarrow{w} q\right) = Z(s, q). \tag{3.327}$$

By Lemma 3.13.5, $\Delta(q)$ will also not improve (change) anymore, which concludes the proof.

■

This captures the intuition of Dijkstra that it progressively "improves" the estimates of $Z\left(s, q\right)$ with $\Delta\left(q\right)$ until $\Delta\left(q\right) = Z\left(s, q\right)$. We can now state the main theorem of correctness for Dijkstra.

Theorem 3.13.1. *Let $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ be an idempotent superior semiring and $\mathcal{G}$ a $\mathbb{K}$-weighted graph. Then, at the end of the execution,* Dijkstra *correctly calculates all single-source distances from the source node s, i.e.:*

$$\Delta\left(q\right) = Z\left(s, q\right) = \bigoplus_{\boldsymbol{\pi} \in \Pi(s,q)} w\left(\boldsymbol{\pi}\right). \tag{3.328}$$

Proof. Before we proceed with the proof, we make an important observation. Due to the idempotency, the pathsum $Z(s, q)$ will always equal the weight of *exactly one* path from s to q—the $\preccurlyeq$-maximally weighted one, since, due to the characterization of $\preccurlyeq$, other paths get absorbed into it. We will use this fact later in the proof. We follow induction on the number of *visited* nodes and prove that the calculated distance is correct for all nodes which have been *visited*, i.e., for all visited q, $\Delta\left(q\right) = \bigoplus\limits_{\boldsymbol{\pi} \in \Pi(s,q)} w\left(\boldsymbol{\pi}\right)$. We will denote the set of vertices in $\mathcal{G}$ with V and the set of vertices already processed as $V \setminus \mathsf{queue}$. **Base case**: Only the source node s has been visited. By the initialization of Δ and $\mathbf{1}$ being the multiplicative unit, the base case holds. **Inductive step**: Assume that the hypothesis holds for the first $n-1$ visited nodes. Let q be the n^{th} processed node and let q'' be any node still in queue. We will show that adding the contribution of paths from s to q going through q'' does not change the pathsum. By definition of $\Delta(q)$ and our inductive hypothesis, we have

$$\Delta(q) = \bigoplus_{q' \in V \setminus \mathsf{queue}} \Delta(q') \otimes w\left(q' \xrightarrow{w} q\right) = \bigoplus_{q' \in V \setminus \mathsf{queue}} Z(s, q') \otimes w\left(q' \xrightarrow{w} q\right). \tag{3.329}$$

By our remark above, this sum is equal to its maximal term, i.e., $\Delta(q) = w(\boldsymbol{\pi}_q)$ for some $\boldsymbol{\pi}_q$ going through a node $q' \in V \setminus \mathsf{queue}$ into q. Suppose that this is not the optimal path from s to q—suppose the optimal path goes through q''. For such a path $\boldsymbol{\pi}$, it would hold that $w(\boldsymbol{\pi}) \prec \Delta(q)$, where $x \prec y \iff x \preccurlyeq y \wedge x \neq y$. Its Δ-value is also equal to some maximal term, for example $\Delta(q'') = w\left(\boldsymbol{\pi}_{q''}\right)$ for some $\boldsymbol{\pi}''$ as above. Furthermore, let r'' be the last node on $\boldsymbol{\pi}_{q''}$ in $V \setminus \mathsf{queue}$ and t'' its child in $\boldsymbol{\pi}_{q''}$. Note that $\Delta(t'') \preccurlyeq \Delta(q)$ since $t'' \in \mathsf{queue}$ and q was chosen over t'' on Line 8 of Alg. 37. We also denote the subpath of $\boldsymbol{\pi}_{q''}$ up to node r'' with $\boldsymbol{\pi}_{r''}$. Then, $\Delta(q) \preccurlyeq w(\boldsymbol{\pi}_{r''}) \otimes w\left(r'' \xrightarrow{w} t''\right)$ Since $r'' \in V \setminus \mathsf{queue}$, by our inductive hypothesis, $\Delta(r'') = Z(s, r'')$. Furthermore, since the subpath $\boldsymbol{\pi}_{r''}$ need not be optimal, we have $w(\boldsymbol{\pi}_{r''}) \preccurlyeq Z(s, r'')$, and lastly, since r'' has been processed, $\Delta(t'')$ has been updated with the weight of the edge between them, i.e., at some point $\Delta(t'') \leftarrow \Delta(t'') \oplus \Delta(r'') \otimes w\left(r'' \xrightarrow{w} t''\right)$. It is easy to see that this means that $\Delta(t'') \preccurlyeq \Delta(r'')$. Combining all of these inequalities together, we get:

$$\begin{aligned}
\Delta(q) &\prec w\left(\boldsymbol{\pi}_{q''}\right) \otimes w\left(q'' \xrightarrow{w} q\right) && \text{(assumption)} && (3.330)\\
&\preccurlyeq w\left(\boldsymbol{\pi}_{r''}\right) \otimes w\left(r'' \xrightarrow{w} t''\right) && && (3.331)\\
&\preccurlyeq \Delta\boldsymbol{\pi}_{r''} \otimes w\left(r'' \xrightarrow{w} t''\right) && && (3.332)\\
&\preccurlyeq \Delta\boldsymbol{\pi}_{r''} && && (3.333)\\
&\preccurlyeq \Delta(q). && && (3.334)
\end{aligned}$$

This implies $\Delta(q) \prec \Delta(q)$, which is a contradiction. Such a path through some q'' therefore cannot exist. This concludes the proof. Note that the proof is identical to the proof of correctness of regular Dijkstra after we have determined that the pathsum only depends on its maximal element. ■

The algorithm uses the data structure *priority queue*, which is an *abstract data type (ADT)*, meaning it is a mathematical model of a structure defined by the operations it supports (queue, dequeue, etc.) but can be implemented in different ways. The correctness of our algorithm does not depend on the underlying implementation of the priority queue, as long as it always returns the minimal element, however, to achieve optimal asymptotic complexity, we usually assume a Fibonacci heap.[33] Using (ordered) lists, its runtime would be quadratic in the number of states, not improving over the runtime of BellmanFord. However, with the appropriate implementation of the priority queue, the bound can be lowered—Dijkstra's algorithm using the Fibonacci attains the lowest known bound for the single-source shortest path problem for graphs with arbitrary non-negative weights. The superiority property ensures it functions correctly on cyclic graphs as well, making Dijkstra very generally applicable. We will soon see an application of Dijkstra as a subroutine in another algorithm, called Johnson's, in **??**.

[33] https://en.wikipedia.org/wiki/Fibonacci_heap

3.14 The Expectation Semiring

3.14.1 Expectation Semirings

This section mostly follows Eisner (2001) and Li and Eisner (2009). This course so far covered many algorithms for manipulating and constructing WFSA. We described the languages they represent, how they can be composed modularly, and how to compute non-trivial operations on the weights. However, we have so far not considered the important question of *where* the weights in a *weighted* finite-state automaton come from. In practice, these weights can either be specified manually, or, more commonly, *learned* so that they fit some sort of a dataset (after the topology of the automaton has been determined). This section focuses on WFSA whose weights represent *probabilities* to allow us to use well known statistical methods.[34] A standard way to fit such statistical models is using the **expectation-maximization (EM)** algorithm. The algorithm fits the parameters of some model to make them compatible with the training data. We will therefore think of automata as being *parametrized* by some parameter vector $\boldsymbol{\theta}$. We will mark that by writing $\mathcal{A}_{\boldsymbol{\theta}}$ to say that the exact specification of the WFSA depends on the parameters $\boldsymbol{\theta}$. In the simplest case, you may think of $\boldsymbol{\theta}$ as simply the weights of the arcs. The framework we will present, however, is more general. $\mathcal{A}_{\boldsymbol{\theta}}$ may, for example, be the result of the composition/intersection of two automata and we may want $\boldsymbol{\theta}$ only to correspond to the weights of the input automata instead of the (many more) weights in the composition. Moreover, we may be interested in parametrizing the WFSA as a log-linear model (for example a CRF introduced in **??**). There, the parameters only implicitly define the weights through the scoring function. As mentioned, such models are then also globally normalized.

We set fit the parameters to be the maximum of some compatibility (objective) function between the parameters and the data. When working with acceptors, the data will come in the form of strings in the language. Our goal then is to set the parameters such that they are compatible with the strings in the dataset. The two most common objectives are the likelihood and the posterior, resulting in the **maximum-likelihood** and **maximum-posterior** estimation. The general EM algorithm can used to solve them.

EM Algorithm The **E** step of the EM algorithm calculates the expected values of the hidden variables in some model. In our case, the hidden variables are the **paths** which generate the strings. Therefore, the E step of the algorithm calculates, for a given fixed parameter $\boldsymbol{\theta}$, which paths are likely to have generated the strings (note that knowing the paths uniquely determines the output string). Then, the **M** step of the algorithm then modifies the parameters $\boldsymbol{\theta}$ to make the paths discovered in the E step more likely.

Example 3.14.1. *To make the idea more concrete, suppose the parameters of the model correspond to the transition probabilities. Then, the E step calculates the expected number of traversals of each edge given the old parameter values and the dataset of accepted strings. The M step then reestimates the probabilities of the transitions from each state to be proportional to the expected number of traversals calculated in the E step.*[35] *If the probabilities are parametrized by a log-linear model, each transition is characterized with a vector of features and these are additively combined along a path (they are "collected" along the path). The E step then calculates the expected vector of total feature counts along the paths yielding the strings in the training data. The M step uses these estimates as*

[34] WFSA whose weights are not explicitly probabilities but arbitrary non-negative numbers can be used too, with some extra work, since they can be either locally/globally normalized.

[35] This is a general idea in the EM algorithm. If we knew the traversal counts exactly, the maximum likelihood estimate of the probabilities would be proportional to these counts. However, since we can only work with estimates, we substitute these counts with the estimates.

actual observed counts and fits the parameters so that the predicted *vector of features by the model matches these "observed" counts.*[36]

First-order Expectation Semiring

Notice that in both cases described in Example 3.14.1, we do some sort of additive grouping of features connected to different transitions along a path. We will exploit this common structure to describe the problem more abstractly as follows. Besides the probability encoded by the transition weights, each path $\boldsymbol{\pi}$ will carry a certain **value** from some set V, which we will denote with $\mathrm{R}(\boldsymbol{\pi})$. The value of a path is the *sum* of the values of the transitions forming the path (in contrast to the weight of the path, which is obtained by multiplying the transition weights). The E step must then calculate the *expected value* of the values of the unknown paths that generated the training dataset. For example, if the value of every transition were just 1, the expected value would be the expected path length. To simplify the notation, we now focus on only 1 training string $\boldsymbol{y}$. Then, the expected value of the path yielding $\boldsymbol{y}$ would be

$$\mathbb{E}\left[\mathrm{R}\left(\boldsymbol{\pi}\right) \mid \boldsymbol{y}\right] = \sum_{\substack{\boldsymbol{\pi}\in\Pi(\mathcal{A})\\ \mathrm{s}(\boldsymbol{\pi})=\boldsymbol{y}}} p\left(\boldsymbol{\pi} \mid \boldsymbol{y}\right)\mathrm{R}\left(\boldsymbol{\pi}\right) \tag{3.335}$$

$$= \sum_{\substack{\boldsymbol{\pi}\in\Pi(\mathcal{A})\\ \mathrm{s}(\boldsymbol{\pi})=\boldsymbol{y}}} \frac{p\left(\boldsymbol{\pi}, \boldsymbol{y}\right)\mathrm{R}\left(\boldsymbol{\pi}\right)}{p\left(\boldsymbol{y}\right)} \tag{3.336}$$

$$= \frac{\sum_{\substack{\boldsymbol{\pi}\in\Pi(\mathcal{A})\\ \mathrm{s}(\boldsymbol{\pi})=\boldsymbol{y}}} p\left(\boldsymbol{\pi}, \boldsymbol{y}\right)\mathrm{R}\left(\boldsymbol{\pi}\right)}{p\left(\boldsymbol{y}\right)}. \tag{3.337}$$

The denominator in Eq. (3.337) is exactly the pathsum of the automaton $\mathcal{A}_{\boldsymbol{\theta}} \otimes \mathcal{A}_{\boldsymbol{y}}$! It turns out, however, that we can also compute the numerator of the expression easily with some bookkeeping. The idea we pursue is to *augment* the weights such that each weight will record both the probability *and* the summary of the parameters that contributed to that probability. This idea is captured with the enforces **invariant**: the weight of any set of paths Π must equal

$$\left(\sum_{\boldsymbol{\pi}\in\Pi} p\left(\boldsymbol{\pi}\right), \sum_{\boldsymbol{\pi}\in\Pi} p\left(\boldsymbol{\pi}\right)\mathrm{R}\left(\boldsymbol{\pi}\right)\right), \tag{3.338}$$

which is sufficient to calculate Eq. (3.337). We see that the weights of the transitions will be tuples, as in the product semiring developed in **??**. To formalize the calculations, we define the **(first-order) expectation semiring** as follows.

Definition 3.14.1 (Expectation Semiring)**.** *The* ***first-order expectation semiring*** *is defined as the tuple* $(\mathbb{R}_+ \times \mathrm{V}, \oplus, \otimes, ^*, \boldsymbol{0}, \boldsymbol{1})$*, where we define*

- $(p_1, v_1) \oplus (p_2, v_2) \overset{\text{def}}{=} (p_1 + p_2, v_1 + v_2)$*;*
- $(p_1, v_1) \otimes (p_2, v_2) \overset{\text{def}}{=} (p_1 p_2, p_1 v_1 + p_2 v_2)$*;*
- $(p, v)^* \overset{\text{def}}{=} (p^*, p^* v p^*)$ *if* p^* *is defined;*
- $\boldsymbol{0} \overset{\text{def}}{=} (0, 0)$*;*

[36] Again, if we had the access to the actual observed counts, we would fit the parameters on those directly.

- $\boldsymbol{1} \overset{\text{def}}{=} (1, 0)$.

You are asked to show that this algebraic structure is indeed a closed semiring in Exercise 3.24. We see that the expectation semiring is thus an instance of a product semiring as introduced in **??**. The first component of the weight tuple keeps the total probability of the set of paths (and is thus a real number) while the second one keeps the expectation of the paths value. The expectation semiring is also closed, which allows us to compute pathsums in arbitrary cyclic machines.

A transition with the probability p and value v is given the weight (p, pv) to satisfy the invariance above for paths of length 0 and 1. The semiring operation definitions are designed to preserve the invariance. This can easily be verified by first showing that $w(\boldsymbol{\pi}) = \bigotimes_{\bullet \xrightarrow{(p,pv)/\bullet} \bullet} (p, pv) = (p(\boldsymbol{\pi}), \mathrm{R}(\boldsymbol{\pi}))$ which can be done by induction on the length of the path. Then, we only have so show that the expectation-semiring-defined addition $\oplus$ correctly combined the weights of a set of paths. $\otimes$ defines the concatenation of paths to get longer paths with the appropriate probability (probabilities of the concatenated paths multiplied together) and appropriate value (the sum of the values of the concatenated paths). $\oplus$ allows us to take the *union* of two disjoint sets of paths and * calculates *infinite* unions.

To compute the expectation Eq. (3.337), we only have to compute the string weight $\boldsymbol{y}$ in the automaton augmented with the expectation-semiring-valued tuple weights. This can be done in the standard way for computing string weights as discussed in **??**.

Example 3.14.2 (Entropy)**.** *A common example of where the expectation semiring pops up is at computing the entropy of the distribution over the paths. There,* $\mathrm{R}\left(\bullet \xrightarrow{\bullet/w} \bullet\right) = -\log w$*. The expression*

$$Z(\mathcal{A}) = \bigoplus_{\boldsymbol{\pi} \in \Pi(\mathcal{A})} w(\boldsymbol{\pi}) \log w(\boldsymbol{\pi}). \tag{3.339}$$

Second-order Expectation Semiring

The idea of first-order expectation semirings can also be extended to computing useful *second-order* statistics, such as the (co)variance and the second-order derivatives (the Hessian). This time, we keep track of *three* quantities per path to satisfy the following invariance:

$$w(\Pi) = \left(\sum_{\boldsymbol{\pi} \in \Pi} p(\boldsymbol{\pi}), \sum_{\boldsymbol{\pi} \in \Pi} p(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi}), \sum_{\boldsymbol{\pi} \in \Pi} p(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi})^\top\right) \tag{3.340}$$

for any set of paths Π. Again, from these, it is easy to construct the variance of R as

$$\mathrm{Cov}(\mathrm{R}(\boldsymbol{\pi})) = \frac{\sum_{\boldsymbol{\pi} \in \Pi} p(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi})^\top - \left(\sum_{\boldsymbol{\pi} \in \Pi} p(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi})\right) \left(\sum_{\boldsymbol{\pi} \in \Pi} p(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi})\right)^\top}{\sum_{\boldsymbol{\pi} \in \Pi} p(\boldsymbol{\pi})}. \tag{3.341}$$

The arc weights are initialized as $\left(p(\boldsymbol{\pi}), p(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi}), p(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi})^\top\right)$. To satisfy the invariance above in the semiring framework, we define **second-order expectation semirings**.

Definition 3.14.2 (Second-order Expectation Semiring)**.** *The* ***first-order expectation semiring*** *is defined as the tuple* $(\mathbb{R}_+ \times \mathrm{V} \times \mathcal{M}(\mathrm{V}), \oplus, \otimes, ^*, \boldsymbol{0}, \boldsymbol{1})$*, where we define*

- $(p_1, v_1, t_1) \oplus (p_2, v_2, t_2) \overset{\text{def}}{=} (p_1 + p_2, v_1 + v_2, t_1 + t_2)$*;*
- $(p_1, v_1, t_1) \otimes (p_2, v_2, t_2) \overset{\text{def}}{=} \left(p_1 p_2, p_1 v_1 + p_2 v_2, p_1 t_1 + p_2 t_2 + 2 v_1 v_2^\top\right)$*;*
- $(p, v, t)^* \overset{\text{def}}{=} \left(p^*, p^* v p^*, p^* p^* \left(2 p^* v v^\top + t\right)\right)$ *if* p^* *is defined;*
- $\boldsymbol{0} \overset{\text{def}}{=} (0, 0, 0)$*;*
- $\boldsymbol{1} \overset{\text{def}}{=} (1, 0, 0)$.

Like in the case of the first-order expectation semiring, the proof that the semiring defined satisfies the invariance boils down to showing that $w(\boldsymbol{\pi}) = \bigotimes_{\bullet \xrightarrow{(p, pv, pvv^\top)/\bullet} \bullet} \left(p, pv, pvv^\top\right) = \left(p(\boldsymbol{\pi}), p(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi}), p(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi}) \mathrm{R}(\boldsymbol{\pi})^\top\right)$ and that the additive operation correctly combined the weights of sets of paths. As before, the calculation of these quantities can be done using any pathsum algorithm on the augmented automaton with the second-order-semiring-valued weights.

Expectation Semirings and Forward-mode Autodifferentiation

The expectation semiring can also be used to compute the *gradients* of the function realized by the automaton with regards to the parameters $\boldsymbol{\theta}$. If we again focus on a single training example $\boldsymbol{y}$, we are interested in $\mathcal{A}_{\boldsymbol{\theta}}(\boldsymbol{y})$ and $\frac{\partial \mathcal{A}_{\boldsymbol{\theta}}(\boldsymbol{y})}{\partial \boldsymbol{\theta}}$. We can calculate both jointly using the first-order semiring by setting the values of the transition weights to $(w, \nabla_{\boldsymbol{\theta}} w)$, where w is the weight of the transition, as parametrized by $\boldsymbol{\theta}$. The operations defined in Def. 3.14.1 remain the same. This works because the definition of the $\otimes$-multiplication in the expectation semiring is essentially the product rule of differentiation. This means that the computation yields $\left(\mathcal{A}_{\boldsymbol{\theta}}(\boldsymbol{y}), \frac{\partial \mathcal{A}_{\boldsymbol{\theta}}(\boldsymbol{y})}{\partial \boldsymbol{\theta}}\right)$.

3.15 The k-best Semiring

3.15.1 The k-best Semiring

Abstract formulation of problems in the semiring framework allows us to solve them efficiently using *dynamic programming* as long as the function we are computing has a suitable formulation in the form of partial solutions. This often allows us to compute operations on exponentially large sets in tractable time. We have seen multiple instances of this so far. For example, we saw that inference and normalization are just two instances of the same pathsum problem over different semirings. However, so far, we have only considered problems returning *exactly one* solution (e.g., the maximal-scoring one or the sum of all scores). Sometimes, we might be interested in returning the (ranked) *set* of solutions satisfying some criterion. This section presents a semiring that allows us to do that. Specifically, we focus on totally ordered semirings. This allows us to rank possible solutions (string sums over some automaton) and return the top k of those. We will see how we can encode this conveniently in with a semiring.

The idea is simple. Distributivity in semirings allows us to act locally when constructing globally optimal solutions—we can be sure that choosing a locally optimal solution *at all* steps of the algorithm results in a globally optimal solution. If we did not choose the optimal option at any point in the algorithm, a better one would exist. We can extend this to a (slightly) suboptimal solution. When constructing the second-best solution, for example, we must have made at least one suboptimal local decision. In fact, the decision must have been optimal in *all but* exactly one of the steps (otherwise a better, yet still suboptimal, solution would exist!). We can use the same reasoning to conclude that the n^{th} best parse must include *at most* $n-1$ suboptimal decisions. Moreover, all but one of these suboptimal decisions must have been included in the previous $n-1$ better solutions by the same reasoning as above. The reason we say *at most* $n-1$ suboptimal decisions is because the suboptimal decisions can be mutually exclusive—for the third best solution, we may make 2 suboptimal decisions at separate steps, or make a single more suboptimal one than in the second best solution at the same step as in the second best one (Charniak and Johnson, 2005). These ideas are captured in the following definition of the **k-best Semiring**.

Definition 3.15.1 (k-best Semiring)**.** *Let $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ be some totally ordered semiring. We define the k-**best semiring** as the tuple $\left(\mathbb{K}^k, \oplus_k, \otimes_k, \boldsymbol{0}_k, \boldsymbol{1}_k\right)$, where, for $x, y \in \mathbb{K}^k$:*

- *$x \oplus_k y = (z_1, \ldots, z_k)$ where $\{z_1, \ldots z_{2k}\} = x \cup y$ and $z_{2k} \preccurlyeq \ldots \preccurlyeq z_1$;*
- *$x \otimes_k y$ proceeds as follows:*
  - *(i) Calculate the k^2 elements $\{x_i \otimes y_j \mid i, j \in [k]\}$;*
  - *(ii) Sort $\{x_i \otimes y_j \mid i, j \in [k]\}$ into $z_{k^2} \preccurlyeq \ldots \preccurlyeq z_1$;*
  - *(iii) Set $x \otimes_k y$ to $(z_1, \ldots, z_k)$;*
- *$\boldsymbol{0}_k = (\boldsymbol{0}, \ldots, \boldsymbol{0})$;*
- *$\boldsymbol{1}_k = (\boldsymbol{1}, \ldots, \boldsymbol{1})$.*

The fact that the above structure forms a semiring is not obvious, but not hard to argue. You are asked to do so in **??**. Thus, running a dynamic program with the above semiring allows us to easily construct the k best solutions to a pathsum problem in an arbitrary totally ordered semiring! The $\oplus_k$ operation allows us to *combine* two sets of solutions and choose the best k of those. This is a generalization of the regular $\oplus$ operation in a semiring that combines two top-1 solutions. The $\otimes_k$

operation similarly generalizes the $\otimes$ operation in the original semiring. $\otimes$ intuitively *combines* two top-1 sub-solutions into a larger one. Due to transitivity, when running a dynamic program, we are allowed to only consider the best-ranking partial solutions. Similarly, when aiming to construct top k solutions, keeping the top k partial solutions suffices to construct the optimal complete solution. However, when combining two partial solutions, we have to consider all k^2 possible ways that these could be combined—all possible top k solutions from the first partial solution could be combined with any of the top k solutions of the second one. This is captured in the 3-step calculation of the $\otimes_k$ operation.

k-best solutions are useful in a variety of use cases. One possible application is optimization with regards to a function f which has no compatible representation in terms of subsolutions and is therefore not suitable for optimization with dynamic programming. Optimizing such a function could be prohibitively expensive on an entire large dataset. We can use a proxy g for f which can be solved with dynamic programming and then optimize f only over the top k solutions found by g. Another instance of where finding k best solutions is useful is in *cascaded optimization.* Modules in Natural Language Processing are often decomposed into cascades of modules (Huang and Chiang, 2005). We may want to optimize the modules' objectives jointly. However, often a module may be incompatible with dynamic programming. Therefore, we may want to postpone some disambiguation by propagating entire k-best lists to subsequent steps in the cascade.

3.16 Johnson's Algorithm

3.17 Exercises

Exercise 3.1

Prove that concatenation, defined as $\boldsymbol{x} \circ \boldsymbol{y} = \boldsymbol{x}\boldsymbol{y}$ for two strings $\boldsymbol{x}$ and $\boldsymbol{y}$, is associative.

Exercise 3.2

As mentioned in §3.1.1, we can simulate ϕ-transitions with alternative ones and result in a ϕ-transition-free automaton. Describe how we can replace the ϕ-transitions in an automaton $\mathcal{A}$.

Exercise 3.3

The set of all formal power series on an alphabet Σ, together with the defined operations of addition and convolution, defines a semiring with an appropriate choice of the units $\mathbf{0}$ and $\mathbf{1}$. Characterize $(\mathcal{W}\langle\langle\Sigma^*\rangle\rangle, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ as a semiring and show that it satisfies the axioms.

Exercise 3.4

Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA, $\boldsymbol{y} \in \Sigma^*$ and $b \in \Sigma$. Show that

$$\boldsymbol{\alpha}_{\mathcal{A}}(\boldsymbol{y} \circ b) = \bigoplus_{q \in Q} \bigoplus_{q \xrightarrow{b/w} q' \in \mathcal{E}(q)} \boldsymbol{\alpha}_{\mathcal{A}}(\boldsymbol{y}, q) \otimes w.$$

Exercise 3.5

Show that the tuple $\mathcal{W} = (\mathbb{R}_+, \max, \cdot, 0, 1)$ indeed defines a semiring.

Exercise 3.6

Show that $(\mathbf{M}^{\otimes n})_{ij}$ encodes the sum of all paths of *exactly* n from i to j in the graph encoded by the matrix $\mathbf{M}$.

Hint: Induction on the path length.

Exercise 3.7

Complete the proof of Prop. 3.6.3 to show that the structure $(\mathbf{M}_D(\mathcal{W}), +, \cdot, \mathbf{O}, \mathbf{I})$ forms a semiring.

Exercise 3.8

Show that for any $\mathbf{M}$, if $\mathbf{M}^* = \mathbf{M}^{(K)}$ for some $K \in \mathbb{N}$ (i.e., the infinite sum is finite), then $\mathbf{M}$ and $\mathbf{M}^*$ $\otimes$-commute, i.e.,

$$\mathbf{M} \otimes \mathbf{M}^* = \mathbf{M}^* \otimes \mathbf{M} \tag{3.342}$$

Exercise 3.9

As mentioned in the discussion of Lehmann's algorithm, the naive implementation is very inefficient with space as it uses D different matrices. Devise a version of the algorithm that only uses 2 matrices during the entire run of the algorithm.

Exercise 3.10

Show that the string semiring as defined in §3.8.1 is in fact a semiring.

Exercise 3.11

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two WFSAs such that at least one of them is acyclic. Show that then $\mathcal{A}_1 \cap \mathcal{A}_2$ is also acyclic.

Exercise 3.12

Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a general WFSA over the semiring $\mathcal{W} = (\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ and $\boldsymbol{y} \in \Sigma^*$. Let $\mathcal{A}_{\boldsymbol{y}}$ be an automaton accepting only the string $\boldsymbol{y}$ with weight $\mathbf{1}$. Show that the automaton $\mathcal{A} \cap \mathcal{A}_{\boldsymbol{y}}$ is acyclic. What algorithm would you, therefore, use to compute its pathsum and with it $\boldsymbol{y}$'s acceptance weight? Considering this, what is the worst-case complexity for computing $\mathcal{A}(\boldsymbol{y})$?

Exercise 3.13

Prove that the product semiring of two semirings as defined in §3.8.1 is in fact a semiring.

Exercise 3.14

Show that $d[q_S, n(e)] = d[q_S, p(e)] \otimes w(e)\ \forall e \in \mathcal{E}(S)$ is equivalent to $d[q_S, n(\boldsymbol{\pi})] = d[q_S, p(\boldsymbol{\pi})] \otimes w(\boldsymbol{\pi})$ for any path $\boldsymbol{\pi}$ in S.

Exercise 3.15

Implement the WFST composition described in §3.8.2 in a function called COMPOSE in the class FST in rayuela (). You can assume that the the transducers are ε-free.

Exercise 3.16

Improve the solution to Exercise 3.15 by adding the handling of ε symbols. **Hint**: The logic is the same as described in the intersection of acceptors in §3.5.

Exercise 3.17

Complete the proof of Thm. 3.11.9, showing that the weight of a path is preserved by weight-pushing.

Solution 3.17

$$\lambda_{\text{PUSH}}\left(p\left(\boldsymbol{\pi}\right)\right) \otimes w_{\text{PUSH}}(\boldsymbol{\pi}) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) \tag{3.343}$$

$$= \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes \Psi(p\left(\boldsymbol{\pi}\right)) \otimes w(\boldsymbol{\pi}) \otimes \Psi(n\left(\boldsymbol{\pi}\right)) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) \qquad \text{(definition)} \tag{3.344}$$

$$= \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes \Psi(p\left(\boldsymbol{\pi}\right)) \otimes \left(\bigotimes_{q_s \xrightarrow{a/w'} q_t \in \boldsymbol{\pi}} \Psi^{-1}(q_s) \otimes w \otimes \Psi(q_t) \right) \tag{3.345}$$

$$\otimes \Psi(n\left(\boldsymbol{\pi}\right)) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) \qquad \text{(definition)} \tag{3.346}$$

$$= \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes \Psi(p\left(\boldsymbol{\pi}\right)) \otimes \Psi^{-1}(q_1) \otimes w_1 \otimes \Psi(q_2) \otimes \Psi^{-1}(q_2) \otimes w_2 \otimes \Psi(q_3) \otimes \tag{3.347}$$

$$\ldots \otimes \Psi^{-1}(q_{n-1}) \otimes w_{n-1} \otimes \Psi(q_n) \otimes \Psi(n\left(\boldsymbol{\pi}\right)) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) \qquad \text{(writing out the } \bigotimes\text{)} \tag{3.348}$$

$$= \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes \Psi(p\left(\boldsymbol{\pi}\right)) \otimes \Psi^{-1}(p\left(\boldsymbol{\pi}\right)) \otimes w_1 \otimes \Psi(q_2) \otimes \Psi^{-1}(q_2) \otimes w_2 \otimes \Psi(q_3) \otimes \tag{3.349}$$

$$\ldots \otimes \Psi^{-1}(q_{n-1}) \otimes w_{n-1} \otimes \Psi(n\left(\boldsymbol{\pi}\right)) \otimes \Psi(n\left(\boldsymbol{\pi}\right)) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) \qquad \text{(definition)} \tag{3.350}$$

$$= \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes \mathbf{1} \otimes w_1 \otimes \mathbf{1} \otimes w_2 \otimes \ldots \otimes w_{n-1} \otimes \mathbf{1} \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) \qquad \text{(definition)} \tag{3.351}$$

$$= \lambda\left(p\left(\boldsymbol{\pi}\right)\right) \otimes w_1 \otimes w_2 \otimes \ldots \otimes w_{n-1} \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) \qquad \text{(neutral element)} \tag{3.352}$$

$$= \lambda\left(p\left(\boldsymbol{\pi}\right)\right) w\left(\boldsymbol{\pi}\right) \otimes \rho\left(n\left(\boldsymbol{\pi}\right)\right) \qquad \text{(definition)} \tag{3.353}$$

Note that any state $q \in Q$ such that $\boldsymbol{\beta}\left(q\right) = \mathbf{0}$ is by definition useless so we do not consider it.

Exercise 3.18

Show that the relation on states defined in Def. 3.11.1 is an equivalence relation, i.e., that it is *also* symmetric and reflexive.

Exercise 3.19

Show that the relation on strings $\sim_L$, defined as $\boldsymbol{x} \sim_L \boldsymbol{y}$ if and only if there is *no* distinguishing extension for $\boldsymbol{x}$ and $\boldsymbol{y}$ is an equivalence relation.

Exercise 3.20

Prove Lemma 3.11.4

Exercise 3.21

Prove Lemma 3.11.5

Exercise 3.22

Prove Lemma 3.11.6

Exercise 3.23

Prove Lemma 3.11.7

Exercise 3.24

Show that the expectation semiring defined in Def. 3.14.1 is indeed a closed semiring.

Exercise 3.25

Show that the k-best semiring defined in Def. 3.15.1 is indeed a semiring.

Index

Bibliography

Cyril Allauzen and Mehryar Mohri. 2003. Efficient algorithms for testing the twins property. *Journal of Automata, Languages and Combinatorics*, 8(2):117–144.

Cyril Allauzen and Mehryar Mohri. 2008. 3-way composition of weighted finite-state transducers. In *International Conference on Implementation and Application of Automata*, pages 262–273. Springer.

Cyril Allauzen, Mehryar Mohri, and Brian Roark. 2003. Generalized algorithms for constructing statistical language models. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 40–47.

Christoph Berkholz, Paul Bonsma, and Martin Grohe. 2017. Tight lower and upper bounds for the complexity of canonical colour refinement. *Theory of Computing Systems*, 60(4):581–614.

Eugene Charniak and Mark Johnson. 2005. Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 173–180, Ann Arbor, Michigan. Association for Computational Linguistics.

Manfred Droste, Werner Kuich, and Heiko Vogler. 2009. *Handbook of Weighted Automata*. Springer Science & Business Media.

Jason Eisner. 2001. Expectation semirings: Flexible em for learning finite-state transducers. In *Proceedings of the ESSLLI workshop on finite-state methods in NLP*, pages 1–5.

Jason Eisner. 2002. An interactive spreadsheet for teaching the forward-backward algorithm. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*, pages 10–18, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Eugene Fink. 1992. *A survey of sequential and systolic algorithms for the algebraic path problem*. Faculty of Mathematics, University of Waterloo.

Michel Fliess. 1974. Matrices de hankel. *Journal de Mathématiques Pures et Appliquées*, 53:197–222.

Michel Gondran and Michel Minoux. 2008. *Graphs, Dioids and Semirings: New Models and Algorithms*, volume 41. Springer Science & Business Media.

John Hopcroft. 1971. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. Automata theory, languages, and computation. *International Edition*, 24(2).

Liang Huang. 2008. Advanced dynamic programming in semiring and hypergraph frameworks. In *Coling 2008: Advanced Dynamic Programming in Computational Linguistics: Theory, Algorithms and Applications - Tutorial notes*, pages 1–18, Manchester, UK. Coling 2008 Organizing Committee.

Liang Huang and David Chiang. 2005. Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 53–64, Vancouver, British Columbia. Association for Computational Linguistics.

Thomas F. Icard. 2017. Beyond almost-sure termination. *Cognitive Science*.

Thomas F. Icard. 2020. Calibrating generative models: The probabilistic chomsky–schützenberger hierarchy. *Journal of Mathematical Psychology*, 95:102308.

Daniel Jurafsky and James H. Martin. 2022. *Speech and Language Processing*, 3 edition. Prentice-Hall, Inc., USA.

Stefan Kiefer. 2020. Notes on equivalence and minimization of weighted automata.

Daniel J. Lehmann. 1977. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76.

Zhifei Li and Jason Eisner. 2009. First-and second-order expectation semirings with applications to minimum-risk training on translation forests. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 40–51.

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

Mehryar Mohri. 2009. *Weighted Automata Algorithms*, pages 213–254. Springer Berlin Heidelberg, Berlin, Heidelberg.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 2008. Speech recognition with weighted finite-state transducers. In *Springer Handbook of Speech Processing*, pages 559–584. Springer.

Mehryar Mohri and Michael Riley. 2001. A weight pushing algorithm for large vocabulary speech recognition. In *Seventh European Conference on Speech Communication and Technology*.

Mehryar Mohri et al. 2002. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350.

Andrew Y Ng. 2003. *Shaping and policy search in reinforcement learning*. University of California, Berkeley.

Robert Paige and Robert E. Tarjan. 1987. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989.

Julien Quint. 2004. On the equivalence of weighted finite-state transducers. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions*, pages 182–185.

Robert Endre Tarjan. 1981. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593.