CHAPTER 1

# Finite-State Machines

This is a book about **weighted finite-state transducers** (WFSTs) and their use in text generation and processing. The WFST formalism synthesizes decades of research into graphs, automata, and formal languages, including lines of research blossoming long before the era of ubiquitous digital computing.

The history of finite-state technology stretches back almost a century. Some key theorems and algorithms were discovered—and rediscovered—long before computers became powerful enough to exploit them (see chapter 5 for an example) and in some cases decades have elapsed between discovery and software implementation. Some essential algorithms were not generalized until the 1990s or later, as part of efforts—particularly at AT&T Bell Labs, and later at Google—to use WFSTs for scalable automatic speech recognition and text-to-speech synthesis.

A few key notions connect these disparate areas of research and application. The first is that of the **state machine**, a sort of abstract mathematical model of computation of which weighted finite-state transducers are a special case. Such models, first formalized by Turing (1936), are not only the foundation of the theory of computation—quite literally, the study of what it means to compute—but also inspired the creation of ENIAC, the first general-purpose digital computer, a decade later. The second is that of **formal languages**. While the origins of formal language theory can be traced at least as far back as Thue (1914), perhaps the most important contribution is a study by Kleene (1956) first circulated in 1951. Kleene's study springs from an obscure goal: the formal characterization of the expressive capacity of "nerve nets", a primitive form of artificial neural network proposed by McCulloch and Pitts (1943) a few years prior. To do so, Kleene introduces a family of formal languages called the "regular languages" and established strong connections between the algebraic characterizations of formal language theory and the automata (i.e., state machine) characterizations used by Turing and others. This body of work was an enormous inspiration in the development of modern linguistic theory—generative grammar in particular (Chomsky 1963)—and also contributed to the theory of compilers, computer programs which translate other computer programs. This chapter traces these two threads—automata and formal languages—and their relationship.

All of this effort, by some of the greatest scientific minds of the early 20th century, could easily have come to naught had the objects of study—regular languages and finite-state automata—turned out to have limited real-world relevance. But it turns out that these exhibit tantalizing similarities to phenomena found in natural—that is, human—languages, a fact which has only become clearer with time. A few examples should suffice. It is now believed that vir-

tually all patterns that define the phonology—or the grapheme-to-phoneme rules—of natural languages can be expressed as relations between regular languages. The hypothesis space of automatic speech recognizers, consisting of a probabilistic mapping between acoustic observations and word sequences, can also be compactly expressed as a relation between two regular languages. Finally, many text generation and processing problems can be framed as transductions between regular languages. Thanks to Kleene and others, it is known that these types of relations can be encoded by state machines, and subsequent work introduces techniques for combining, applying, optimizing, and searching these machines.

## 1.1 STATE MACHINES

A **state machine** is hardware or software whose behavior can be described solely in terms of a set of **states** and **arcs**, which represent transitions between those states. In this formalism, states roughly correspond to "memory" and arcs to "operations" or "computations". State machines are examples of what computer scientists call **directed graphs**.[1] These are "directed" in the sense that the existence of an arc from state $q$ to state $r$ does not imply an arc from $r$ to $q$. A **finite-state machine** is merely a state machine with a finite, predetermined set of states and labeled arcs.

One familiar example of a state machine—encoded in hardware, rather than software—is the old-fashioned gumball machine (Figure 1.1). Such machines can be in exactly one of two states at a time, and each state is associated with actions such as

- turning the knob,
- inserting a coin, or
- emitting a gumball.

At one state, arbitrarily called state 0, it is possible to turn the knob, but this has no effect on the behavior of the machine. If, on the other hand, one inserts the appropriate coin(s), that transitions the machine to a state 1, at which point a subsequent turn of the knob will cause the machine to emit a gumball and return to state 0. This of course is an idealization of real-world gumball machines, which may experience mechanical failure or run out of gumballs. Without a shop-keeper around to service the machine, model and reality necessarily diverge.

The description of the gumball machine above is given a graphical representation in Figure 1.2. By convention, the bold outline of state 0 indicates that it has been—arbitrarily—chosen as the **start** or **initial state**; the double-struck outline indicates that it is also a **final state**; these notions will be formalized shortly. Valid transitions between states are indicated with arrows. These arcs are labeled with pairs of actions. Here, the inputs are user actions and the outputs are gumballs. The Greek letter $\epsilon$ ("epsilon") is used to represent the absence of an input and/or

[1] The primary difference is terminological; what are here called **states** and **arcs** are known in other communities as "vertices" and "edges", respectively.

Figure 1.1: An old-fashioned gumball machine. (Image credit: Dario Lo Presti/Shutterstock.com)

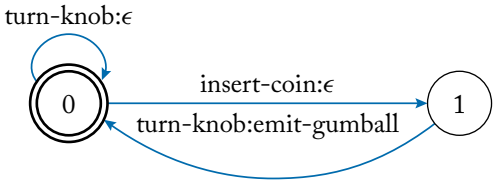


Figure 1.2: An old-fashioned gumball machine schematized as a state machine.

output for a given arc. Because, as mentioned, turning the knob at state 0 produces no output and does not change the state of the machine, there is a self-arc at state 0 labeled `turn-knob:`$\epsilon$. On the other hand, inserting a coin at state 0 produces no observable output, but it transitions the machine to state 1. At this state a knob turn by the user causes the machine to emit a gumball and return to state 0.

We now provide definitions for various types of finite-state machine, after reviewing some formal preliminaries.

## 1.2 FORMAL PRELIMINARIES

This section provides a brief introduction to set theory and related topics. Those readers already familiar with sets, relations, functions, strings, and languages are welcome to skip to section 1.3.

### 1.2.1 SETS

**Sets** are abstract, unordered collections of distinct objects. They are an abstract, purely logical notion, and their definition does not presuppose any particular method of representing them in hardware or software; they are unordered in the sense that there is no natural ordering among the **elements** or **members** of any set. By convention, sets are represented using uppercase Greek or Italic letters, and elements of sets are denoted using lowercase Italic letters. Set membership is indicated using the $\in$ symbol, e.g., $x \in X$ is read "$x$ is a member of $X$". Non-membership is written using the $\notin$ symbol, e.g., $x \notin X$ is read "$x$ is not a member of $X$".

Members of a set can be any type of object, including other sets. There are several ways to specify the members of a set. First, for finite sets, one can simply list the elements in the set enclosed in curly braces, a representation called **extensional** or **list notation**. For instance, $\{2, 3, 5, 7\}$ is the finite set of prime numbers less than 10. An alternative notation, and the only one which can be used to denote infinite sets, uses a predicate such that if some element satisfies the predicate, that element is a member of a set; this is known as **intensional**, **predicate**, **set-builder**, or **set-former** notation. For instance, one might indicate the infinite set of prime numbers using the notation $\{x \mid \text{prime}(x)\}$. Finally, special notation is used for the **empty set**, the set with no elements: it is written $\emptyset$. The **cardinality** of a set $X$, written $|X|$, is the number of distinct elements in the set.

A set $X$ is said to be a **subset** of another set $Y$ if every element in $X$ is also a member of $Y$. This property is written using the $\subseteq$ operator, e.g., $X \subseteq Y$ is read "$X$ is a subset of $Y$". $X$ is a **proper subset** of $Y$ ($X \subset Y$) if and only if $X$ is a subset of $Y$ and $X \neq Y$.

There are various logical operations over sets. Given two sets $X$ and $Y$, their **intersection** $X \cap Y$ is the set that contains all elements which are members of both $X$ and $Y$: that is, $X \cap Y = \{z \mid z \in X \wedge z \in Y\}$ where $\wedge$ represents logical AND. Given two sets $X$ and $Y$, their **union** $X \cup Y$ is the set that contains all elements which are members of $X$, $Y$, or both: that is, $X \cup Y = \{z \mid z \in X \vee z \in Y\}$ where $\vee$ represents logical OR. Finally, their **difference** $X - Y$ is the set that contains all elements which are members of $X$ but not of $Y$: that is, $X - Y = \{z \mid z \in X \wedge z \notin Y\}$.

### 1.2.2 RELATIONS AND FUNCTIONS

A **pair** or **two-tuple** is a sequence of two elements, e.g., $(a, b)$ is the pair consisting of $a$ then $b$. This is used to define an operation over sets known as the **cross-product** or **Cartesian product**. Given two sets $X$ and $Y$, their cross-product $X \times Y$ is the set containing all ordered pairs $(x, y)$ where $x$ is an element of $X$ and $Y$ is an element of $Y$. That is, $X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$.

A **relation**—specifically, a **binary** or **two-way relation**—over sets $X$ and $Y$ is a subset of the cross-product $X \times Y$. In this book, relations are indicated using lowercase Greek letters, and the **domain**—set of inputs—and **range** (or more properly, the **codomain**)—the set of outputs—are usually provided upon first definition. For instance, the expression $\gamma \subseteq X \times Y$ indicate that $\gamma$ is a relation with domain $X$ and range $Y$. Relations represent mappings between elements of the domain and elements of the range; for instance, the "less than" relation can be written $\lambda \subseteq \mathbb{R} \times \mathbb{R} = \{(x, y) \mid x < y\}$ where $\mathbb{R}$ is the set of real numbers.

A **function** is a relation for which every element of the domain is associated with exactly one element of the range. The "less than" relation above is not a function because, for example, there are an infinitude of real numbers that are less than any other real number. However, the "successor" relation $\sigma \subseteq \mathbb{N} \times \mathbb{N} = \{(x, x + 1) \mid x \in \mathbb{N}\}$, where $\mathbb{N}$ is the set of natural numbers, is a function, because each natural number has exactly one successor.

Three-, four-, and five-way relations, and so on, are all well-defined, though there is no such generalization for functions, since $n$-way relations where $n > 2$ lack well-defined domain and range. However, one can redefine any $n$-way relation into a two-way relation by grouping the various sets into domain and range; for instance, a four-way relation over $A \times B \times C \times D$ can be redefined as a two-way relation (and possibly, a function) with domain $A \times B$ and range $C \times D$. Such a relation might be defined as a subset of $A \times B \to C \times D$, with the arrow used to indicate the partition into domain and range.

The application of an input argument to a relation or function can be indicated using square brackets. For instance, given the successor function $\sigma$, then $\sigma[3] = \{4\}$ because $(3, 4) \in \sigma$.

Given a relation $\gamma \subseteq X \times Y$ and $x \in X$, $\gamma[x] \downarrow$ indicates that $\gamma$ is well defined at $x$ and $\gamma[x] \uparrow$ indicates that $\gamma$ is undefined at $x$. A relation or function is said to be **total** if it is defined for all values of the domain. The less-than relation and successor functions, for example, are both total.

### 1.2.3 STRINGS AND LANGUAGES

Many of the sets defined below contain a type of element known as a string. Let $\Sigma$ be a set of symbols called the **alphabet**. A **string** is a finite ordered sequence of zero or more elements from the alphabet. By convention, the empty string is indicated by $\epsilon$. Note that $\epsilon$ is not a member of $\Sigma$.

The **concatenation** of two strings is the string produced by joining the two strings end-to-end. The concatenation of two strings $x, y$ is written $xy$. Note that $\epsilon$ is the concatenative identity, thus $x\epsilon = \epsilon x = x$ for all $x$.

A set of zero or more strings is known as a **language**.[2] Since languages are sets, operations such as intersection, union, and difference are well defined. In addition, concatenation can also be generalized to languages, i.e., given languages $X$ and $Y$, $XY = \{xy \mid x \in X \wedge y \in Y\}$. One other operation over languages is closure. First, the notation $X^n$, where $n$ is a natural number, denotes a language consisting of $n$ self-concatenations of $X$; e.g., $X^0 = \{\epsilon\}$ and $X^4 = XXXX$. The (**concatenative**) **closure** of a language $X$ is an infinite union of zero or more concatenations of $X$ with itself. It is notated with a superscripted asterisk, e.g., $X^* = \bigcup_{i \geq 0} X^i = \{\epsilon\} \cup X \cup XX \cup XXX \cup \ldots$. One variant of closure, indicated with a superscript plus-sign, excludes the empty string, e.g., $X^+ = \bigcup_{i > 0} X^i = X \cup XX \cup XXX \cup \ldots$, or equivalently, $X^+ = XX^*$. These two variants of closure are sometimes referred to as **Kleene star** and **Kleene plus**, respectively. Finally, a superscripted question mark is used to indicate optionality, e.g., $X^? = \{\epsilon\} \cup X$.

## 1.3 ACCEPTORS AND REGULAR LANGUAGES

Finite acceptors are the simplest form of finite automata, in some ways simpler than the model of a gumball machine presented above. They represent a family of string sets known as the regular languages.

### 1.3.1 FINITE-STATE ACCEPTORS

A **finite-state acceptor** (FSA) is a five-tuple consisting of

1. a finite set of states $Q$,
2. a **start** or **initial state** $s \in Q$,
3. a set of **final** (or **accepting**) **states** $F \subseteq Q$,
4. an **alphabet** $\Sigma$, and
5. a **transition relation** $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$.

Note that as formalized here, there is only one start state but there may be many final states; also note that the start state may itself be a final state.[3]

An FSA is said to accept a string if there exists a path from the initial state to some final state, and the labels of the arcs traversed by that path correspond to the string in question. The set of all strings so accepted by an FSA is called its language. More formally, given two states $q, r \in Q$ and a symbol $z \in \Sigma \cup \{\epsilon\}$, $(q, z, r) \in \delta$ implies that there is an arc from state $q$ to state $r$ with label $z$. A **path** through a finite acceptor is a pair of

[2] This is not intended to supplant common-sense notions of what a language is; it is merely a term of art.

[3] One could allow for arbitrarily many start states, but given any finite automaton with multiple start states $S \subseteq Q$, it is trivial to construct an equivalent automaton with a single "superinitial" start state. Alternatively, one could limit the formalism to a single "superfinal" final state $f \in Q$.

1. a state sequence $q_1, q_2, \ldots, q_n \in Q^n$ and a
2. a string $z_1, z_2, \ldots, z_n \in (\Sigma \cup \{\epsilon\})^n$,

subject to the constraint that $\forall i \in [1, n] : (q_i, z_i, q_{i+1}) \in \delta$; that is, there exists an arc from $q_i$ to $q_{i+1}$ labeled $z_i$. A path that visits a state more than one time—i.e., if its state sequence contains the start state $s$ or any repeated states—has a **cycle**. Automata are **cyclic** if any of their paths contain cycles and **acyclic** otherwise.

A path is said to be **complete** if

1. $(s, z_1, q_1) \in \delta$ and
2. $q_n \in F$.

That is, a complete path must also begin with an arc from the initial state $s$ to $q_1$ labeled $z_1$ and terminate at a final state. Henceforth, without loss of generality, $\epsilon$-labels are omitted from path strings because $\epsilon$ signals the absence of a symbol and therefore can be ignored. Indeed, for every FSA, there is an equivalent $\epsilon$-**free** FSA, i.e., an FSA which accepts the same language but which has no $\epsilon$-arcs, computed with the $\epsilon$-removal algorithm (Mohri 2002a). Then, an FSA **accepts** or **recognizes** a string $z \in \Sigma^*$ if there exists a complete path with string $z$. The set of strings accepted by an FSA is called its language.

### 1.3.2 REGULAR LANGUAGES

The family of languages recognized by finite acceptors are the **regular languages**. Kleene (1956) provides an algebraic characterization. Given an alphabet $\Sigma$:

1. The empty language $\emptyset$ is a regular language.
2. The empty string language $\{\epsilon\}$ is a regular language.
3. If $s \in \Sigma$, then the singleton language $\{s\}$ is a regular language.
4. If $X$ is a regular language, then its closure $X^*$ is a regular language.
5. If $X, Y$ are regular languages, then:
   - their concatenation $XY$ is a regular language, and
   - their union $X \cup Y$ is a regular language.
6. Languages which cannot be derived as above are not regular languages.

Kleene (ibid.) also shows that every finite acceptor corresponds to a regular language and that every regular language corresponds to a finite acceptor. This result, known as **Kleene's theorem**, implies that operations over languages such as closure, concatenation, and union are defined not only for languages but also for finite acceptors.

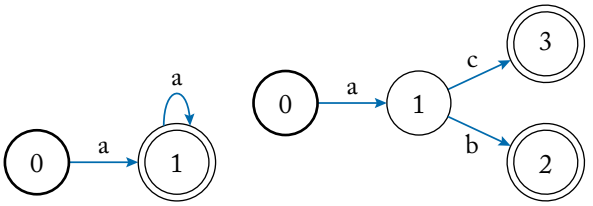


Figure 1.3: Finite acceptors for the languages $\{\mathtt{a}\}^+$ (left) and $\{\mathtt{a}\}(\{\mathtt{b}\} \cup \{\mathtt{c}\})$ (right).

Two examples of FSAs and their corresponding regular languages are shown in Figure 1.3 as **state transition diagrams**. The left pane contains an FSA defined by $Q = \{0, 1\}$, $s = 0$, $F = \{1\}$, $\Sigma = \{\mathtt{a}\}$, and $\delta = \{(0, \mathtt{a}, 1), (1, \mathtt{a}, 1)\}$, which accepts the infinite language $\{\mathtt{a}\}^+ = \{\mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \ldots\}$. The right pane shows an FSA which accepts the finite language $\{\mathtt{a}\}(\{\mathtt{b}\} \cup \{\mathtt{c}\}) = \{\mathtt{ab}, \mathtt{ac}\}$. The reader is encouraged to study these acceptors and manually trace the generation of a few strings.

### 1.3.3 REGULAR EXPRESSIONS

**Regular expressions** are a declarative notational scheme used to characterize the regular languages (Hopcroft et al. 2008: ch. 3). One can convert any finite acceptor to a regular expression, and any regular expression to a finite automaton. However, implementations of regular expressions in many programming languages—for instance, the implementation used in Python's built-in `re` module—include additional features which cannot be encoded using regular languages or finite-state acceptors.

## 1.4 TRANSDUCERS AND RATIONAL RELATIONS

Finite transducers are a generalization of finite acceptors. Rather than modeling languages, they model **rational relations** between pairs of languages, and as such they can be used to encode string-to-string transductions.[4]

### 1.4.1 FINITE-STATE TRANSDUCERS

A **finite-state transducer** (FST) is a six-tuple consisting of

1. a finite set of states $Q$,
2. a start state $s \in Q$,
3. a set of final states $F \subseteq Q$,

[4]It is possible to generalize rational relations, and finite-state transducers, to relations between sets of more than two languages. This generalization is not discussed here as it is only rarely employed in computational linguistics, but see, e.g., Kay 1987, Kiraz 2001, or Hulden 2017.

4. an **input alphabet** $\Sigma$,
5. an **output alphabet** $\Phi$, and
6. a transition relation $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Phi \cup \{\epsilon\}) \times Q$.

The first three elements are also used in the definition of FSAs; the latter three are novel. The key distinction between FSAs and FSTs is that in the latter case, arcs bear pairs of labels, one drawn from an input alphabet and the other from a (possibly disjoint) output alphabet. A **path** through a finite transducer is a triple consisting of

1. a state sequence $q_1, q_2, \ldots, q_n \in Q^n$,
2. an input string $x_1, x_2, \ldots, x_n \in (\Sigma \cup \{\epsilon\})^n$, and
3. an output string $y_1, y_2, \ldots, y_n \in (\Phi \cup \{\epsilon\})^n$,

subject to the constraint that $\forall i \in [1, n] : (q_i, x_i, y_i, q_{i+1}) \in \delta$. A **complete path** is a path where

1. $(s, x_1, y_1, q_1) \in \delta$ and
2. $q_n \in F$.

That is, a complete path must also begin with a transition from the initial state $s$ to $q_i$ with input label $x_i$ and output label $y_i$ and halt in a final state. Without loss of generality, and once again ignoring the presence of $\epsilon$, the domain $\Sigma^*$ and range $\Phi^*$ of an FST are both themselves regular languages, and the FST itself can be interpreted as a relation, a subset of the cross-product $\Sigma^* \times \Phi^*$. Then, an FST **transduces** or **maps** from $x \in \Sigma^*$ to $y \in \Phi^*$ so long as a complete path with input string $x$ and output string $y$ exists. However, unlike FSAs, not all FSTs have an equivalent $\epsilon$-free form. For example, consider an FST mapping from two-character U.S. state abbreviations (e.g., `OH`) to state names (`Ohio`); a fragment of such an FST is shown in Figure 1.4. Here, arcs with $\epsilon$ input labels are necessary to allow input strings which are shorter than the corresponding output strings. Note also that the $\epsilon$-removal algorithm mentioned in subsection 1.3.1 removes $\epsilon$-arcs—those which have $\epsilon$ as both input and output labels—not $\epsilon$-labels in general.

### 1.4.2 RATIONAL RELATIONS

The family of string relations that can be encoded as a finite-state transducer are the **rational relations**. Like regular languages, closure, concatenation, and union are all well defined for rational relations. The rational relations are closed under these operations, meaning that the closure of a rational relation, or the concatenation or union of two or more rational relations, are also rational relations. However, there are other operations, such as difference, under which the regular languages are closed but the rational relations are not.

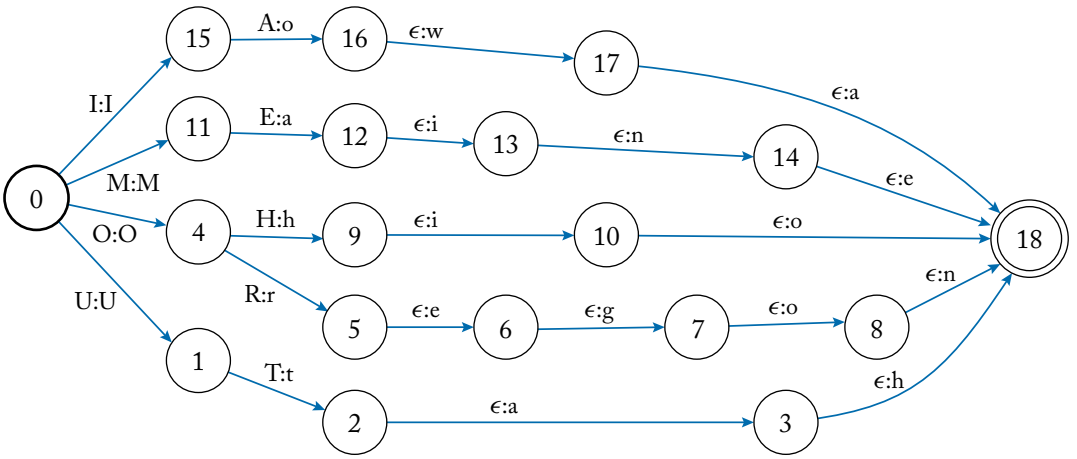


Figure 1.4: Fragment of a FST mapping from state abbreviations to state names.

Rational relations are closely related to, but distinct from, **regular expression substitutions** (e.g., as performed by Python's `re.sub` function).[5] On one dimension, regular expression substitutions are less expressive than rational relations, because the former permit many-to-many (rather than merely one-to-one and many-to-one) transductions, whereas the pattern matched by a `re.sub` is an arbitrary regular language, the substitution must be a single string. Neither finite state transducers nor the rational relations are restricted in this fashion. At the same time, `re.sub` implements other mechanisms that make it more expressive than rational relations.

## 1.5 WEIGHTED ACCEPTORS AND LANGUAGES

The above formalisms also permit an extension in which acceptors and transducers—and languages and relations—are generalized by attaching weights to states and arcs. These weights can represent virtually any set so long as the set and associated operations obey certain constraints decribed below. **Language models**, probability distributions over strings, can be compactly encoded as weighted acceptors (e.g., Allauzen et al. 2003, 2005, Roark et al. 2012); **hidden Markov models** can be encoded as weighted transducers (Roche and Schabes 1995) as can sequential **linear models** (Wu et al. 2014) and decoder graphs for automatic speech recognition engines (e.g., Mohri 1997, Mohri et al. 2002). Below, semirings are defined and exemplified and then used to generalize earlier definitions of automata, languages, and relations.

### 1.5.1 MONOIDS AND SEMIRINGS

Weighted automata algorithms are defined with respect to an algebraic system known as a semiring (Kuich and Salomaa 1986). It is first necessary to define a related notion, monoids.

[5] https://docs.python.org/3/library/re.html#re.sub

A **monoid** is an ordered pair $(\mathbb{K}, \bullet)$ where $\mathbb{K}$ is a set and $\bullet$ is a binary operator over $\mathbb{K}$ with the properties of

1. **closure**: $\forall a, b \in \mathbb{K} : a \bullet b \in \mathbb{K}$,
2. **associativity**: $\forall a, b, c \in \mathbb{K} : (a \bullet b) \bullet c = a \bullet (b \bullet c)$, and
3. **identity**: $\exists e \in \mathbb{K} : e \bullet a = a \bullet e = a$.

A monoid is said to be **commutative** if $\forall a, b \in \mathbb{K} : a \bullet b = b \bullet a$. Then, a **semiring** is then a five-tuple $(\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1})$ such that

1. the pair $(\mathbb{K}, \oplus)$ form a commutative monoid with identity element $\bar{0}$,
2. the pair $(\mathbb{K}, \otimes)$ form a monoid with identity element $\bar{1}$,
3. $\forall a, b, c \in \mathbb{K} : a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$, and
4. $\forall a \in \mathbb{K} : a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$.

These constraints require that $\oplus$ is commutative, that $\bar{0}$ is the additive identity, that $\bar{1}$ is the multiplicative identity, that $\otimes$ distributes over $\oplus$, and that $\bar{0}$ is the multiplicative annihilator (i.e., that any weight multiplied with $\bar{0}$ is $\bar{0}$). Some common semirings are shown in Table 1.1. The **Boolean semiring** consists of true (1) and false (0) values and logical OR and AND operators. The **probability semiring** ranges over positive real numbers $\mathbb{R}_+$ and employs the expected $+$ and $\times$ arithmetic operations for $\oplus$ and $\otimes$.[6] The **log semiring** is the projection of the probability semiring onto the log domain.[7] The log semiring uses the logarithmic identity $\ln(xy) = \ln x + \ln y$ to replace multiplication with addition in the log domain; this helps to avoid arithmetic underflow when weight computations are performed with floating-point numbers. The definition of addition in this semiring is somewhat more complex: $\oplus = \oplus_{\log}$ where $a \oplus_{\log} b = -\ln(e^{-a} + e^{-b})$. Finally, the **tropical semiring** is identical to the log semiring except that $\oplus = \min$.[8]

A semiring is said to exhibit the **path property** (or to be a **path semiring**) if for all $a, b \in \mathbb{K} : a \oplus b \in \{a, b\}$. The tropical semiring has this property—the minimum of any two numbers must be one of those two numbers—as does the Boolean semiring. Non-path semirings such as the probability semiring and log semirings define $\oplus$ in a way consistent with common-sense arithmetic notions, making them suitable for applications that involve counting. One example of this is the expectation maximization algorithm, commonly used to learn free parameters of speech models. In contrast, path semirings are used for decoding because the path property is required to efficiently compute the shortest path(s) through weighted automata (section 4.3).

[6] For probabilities, only numbers between 0 and 1 inclusive make sense, but numbers in the range $(1, +\infty]$ serve as inverse elements.

[7] The OpenFst library uses the natural logarithm, specifically.

[8] The tropical semiring is named in tribute to the late mathematician Imre Simon of the University of São Paulo. We note that São Paulo is just south of the Tropic of Capricorn, so "subtropical" would have been more apt.

Table 1.1: Some commonly used semirings for finite-state applications; $\mathbb{R}$ and $\mathbb{R}_+$ denote the real, and positive real, numbers, respectively.

| | $\mathbb{K}$ | $\oplus$ | $\otimes$ | $\bar{0}$ | $\bar{1}$ |
|---|---|---|---|---|---|
| Boolean | $\{0, 1\}$ | $\vee$ | $\wedge$ | 0 | 1 |
| Probability | $\mathbb{R}_+$ | $+$ | $\times$ | 0 | 1 |
| Log | $\mathbb{R} \cup \{\pm\infty\}$ | $\oplus_{\log}$ | $+$ | $+\infty$ | 0 |
| Tropical | $\mathbb{R} \cup \{\pm\infty\}$ | min | $+$ | $+\infty$ | 0 |

### 1.5.2 WEIGHTED FINITE ACCEPTORS

A **weighted finite-state acceptor** (WFSA) is an FSA in which weights are associated with arcs and states. It is defined by a six-tuple consisting of

1. a finite set of states $Q$,
2. a start state $s \in Q$,
3. a **semiring** $(\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1})$,
4. a **final weight function** $\omega \subseteq Q \times \mathbb{K}$,
5. an alphabet $\Sigma$, and
6. a **transition relation** $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \mathbb{K} \times Q$.

Three modifications have been made with respect to the earlier definition of FSAs in subsection 1.3.1 above. First, WFSAs are defined with respect to a particular semiring. Second, in place of the finite state set $F$ there is a function $\omega$ which gives the **final weight** for each state. By convention, this is assumed to be a total function and a state $q \in Q$ is said to be non-final if $\omega(q) = \bar{0}$.[9] Third, the transition relation $\delta$ has been extended to include weights. A **path** through a weighted finite acceptor is a triple of

1. a state sequence $q_1, q_2, \ldots, q_n \in Q^n$,
2. a string $z_1, z_2, \ldots, z_n \in (\Sigma \cup \{\epsilon\})^n$, and
3. a weight sequence $k_1, k_2, \ldots, k_n \in \mathbb{K}^n$

subject to the constraint that $\forall i \in [1, n] : (q_i, z_i, k_i, q_{i+1}) \in \delta$. This constraint holds that there exists an arc from $q_i$ to $q_{i+1}$ that has the label $z_i$ and weight $k_i$. A path is **complete** if

[9] Alternatively, one could define $\omega$ as a partial function in which $\omega[q] \downarrow$ if and only if state $q$ is final.

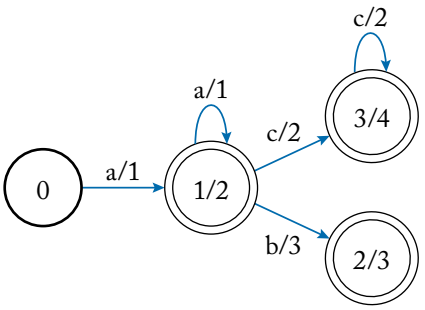


Figure 1.5: Weighted finite acceptor over the language $\{\mathtt{a}\}^+(\{\mathtt{b}\} \cup \{\mathtt{c}\}^*)$.

1. $(s, z_1, k_1, q_1) \in \delta$ and
2. $\omega[q_n] \neq \bar{0}$.

That is, a complete path must also begin with an arc from the initial state $s$ to $q_1$ with label $z_1$ and weight $k_1$ and halt in a final state, i.e., a state with a non-$\bar{0}$ final weight. Once again ignoring $\epsilon$-labels, a WFSA accepts a string $z \in \Sigma^*$ with weight

$$\left(\bigotimes_{i=1}^{n} k_i\right) \otimes \omega[q_n] = k_1 \otimes k_2 \otimes \ldots \otimes k_n \otimes \omega[q_n],$$

if there exists a complete path with string $z$ and weight sequence $k_1, k_2, \ldots, k_n$. Note that the **path weight**, the weight associated with a path, is given by the $\otimes$-product of the weight sequence and the final weight of the final state in the path.

An example WFSA is shown in Figure 1.5; weights are separated from arc and/or state labels by a forward slash. This WFSA accepts the string `aacc`, for example, with weight $1 \otimes 1 \otimes 2 \otimes 2 \otimes 4$, equal to 10 in the log and tropical semirings.

### 1.5.3 WEIGHTED REGULAR LANGUAGES

There are two roughly equivalent ways to define the **weighted regular languages** expressed by weighted finite acceptors. Under one definition, a weighted language is a partial relation over $\Sigma^* \times \mathbb{K}$; that is, it assigns weights to those strings in its language. However, one can alternatively define weighted languages as a total relation with $\bar{0}$ used as the weight for strings not accepted under the previous definition. This eliminates the distinction between those strings not accepted by the language and those accepted with weight $\bar{0}$.

## 1.6 WEIGHTED TRANSDUCERS AND RELATIONS

Finite transducers and relations can also be extended to support weights.

### 1.6.1 WEIGHTED FINITE TRANSDUCERS

The definition of a **weighted finite-state transducer** (WFST) should be obvious from the preceding discussion, but is provided for completeness. A WFST is a seven-tuple consisting of

1. a finite set of states $Q$,
2. a start state $s \in Q$,
3. a semiring $(\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1})$,
4. a final weight function $\omega \subseteq Q \times \mathbb{K}$,
5. an input alphabet $\Sigma$,
6. an output alphabet $\Phi$, and
7. a transition relation $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Phi \cup \{\epsilon\}) \times \mathbb{K} \times Q$.

Paths through a WFST are then four-tuples consisting of

1. a state sequence $q_1, q_2, \ldots q_n \in Q^n$,
2. a input string $x_1, x_2, \ldots, x_n \in (\Sigma \cup \{\epsilon\})^n$,
3. a output string $y_1, y_2, \ldots, y_n \in (\Phi \cup \{\epsilon\})^n$, and
4. a weight sequence $k_1, k_1, \ldots, k_n \in \mathbb{K}^n$

subject to the constraint that $\forall i \in [1, n] : (q_1, x_i, y_i, k_i, q_{i+1}) \in \delta$. A **complete path** is a path where

1. $(s, x_1, y_1, k_1, q_1) \in \delta$ and
2. $\omega[q_n] \neq \bar{0}$.

That is, a complete path must also begin with a transition from the initial state $s$ to $q_1$ with input label $x_1$, output label $y_1$, and weight $k_1$, and halt in a final state. Once again, ignoring the presence of $\epsilon$-labels in the input and output strings, a WFST **transduces** or **maps** from $x \in \Sigma^*$ to $y \in \Phi^*$ with weight $k \in \mathbb{K}$ so long as a complete path with path weight $k$, input string $x$, and output string $y$ exists.

### 1.6.2 WEIGHTED RATIONAL RELATIONS

Each WFST corresponds to a **weighted rational relation**, a three-way partial relation over $\Sigma^* \times \Phi^* \times \mathbb{K}$, but in practice, such relations are often reinterpreted as two-way partial relations over $\Sigma^* \to \Phi^* \times \mathbb{K}$; that is, for a given input string, they yield pairs of an output string and an associated path weight. Weighted relations can alternatively be defined as total relations similarly to the alternative definition of weighted languages given in subsection 1.5.3.

## FURTHER READING

Partee et al. (1993: ch. 1–3) give a gentle introduction to sets, pairs, relations, functions, and strings.

Hopcroft et al. (2008: ch. 2) formalizes finite acceptors, though they eschew both transducers and weights.

Comparable formalizations of WFSTs are given by Roark and Sproat (2007: ch. 1) and Mohri (2009).

Hopcroft et al. (2008: ch. 3) formalize connections between finite acceptors, regular languages, and regular expressions. Jurafsky and Martin (2009: ch. 2) and Eisenstein (2019: ch. 9) briefly discuss these connections.

Hopcroft et al. (2008: ch. 5–7) and Allauzen and Riley (2012) present an extension of finite automata known as **pushdown automata**, corresponding to the family of formal languages known as **context-free grammars** (Chomsky 1963).