Finite-state acceptors

LING83800

Outline

- Motivations
- State machines
- Formalization
- OpenFst and friends
- Demo

Motivations

Kleene 1956

Initially, the study of

- abstract computational devices known as *state machines* and
- formal languages

were considered independent of one another. Kleene (1956) was one of the first to unify these two areas of study. Kleene wished to characterize the properties of *nerve nets* (McCulloch and Pitts, 1943), a primitive form of artificial neural network. In doing so, Kleene introduced the regular languages and established strong connections between regular languages and the *finite acceptors*, a type of state machine.

Regular languages in the 20th century

- Regular languages were popularized in part by discussion of the *Chomsky*(*-Schützenberger*) hierarchy (e.g., Chomsky and Miller, 1963).
- Regular languages were used by Thompson (1968) to create the `grep` regular expression matching utility.
- Finite acceptors are used to compactly store morphological dictionaries.
- Finite acceptors are used to compactly represent *language models*, particularly in speech recognition engines.

It now seems that an enormous amount of linguistically-interesting phenomena can be described in terms of regular languages (and the closely-related *rational relations*, which we'll review next week).

Negative results

At the same time, there were two important negative results:

- Syntactic grammars belong to a higher-classes of formal languages, the *mildly context-sensitive languages* (Vijay-Shanker et al., 1987).
- The class of regular languages are not “learnable” from positive data under Gold’s (1967) notion of *language identification in the limit*.

In practice, this means that regular languages and finite acceptors are somewhat limited as models of syntax, though they are still well-suited as models of phonology and morphology.

Introducing state machines

State machines

A *state machine* is hardware or software whose behavior can be described solely in terms of a set of *states* and *arcs*, transitions between those states. In this formalism, states roughly correspond to “memory” and arcs to “operations” or “computations”. A *finite-state machine* is merely a state machine with a finite, predetermined set of states and labeled arcs.

As directed graphs

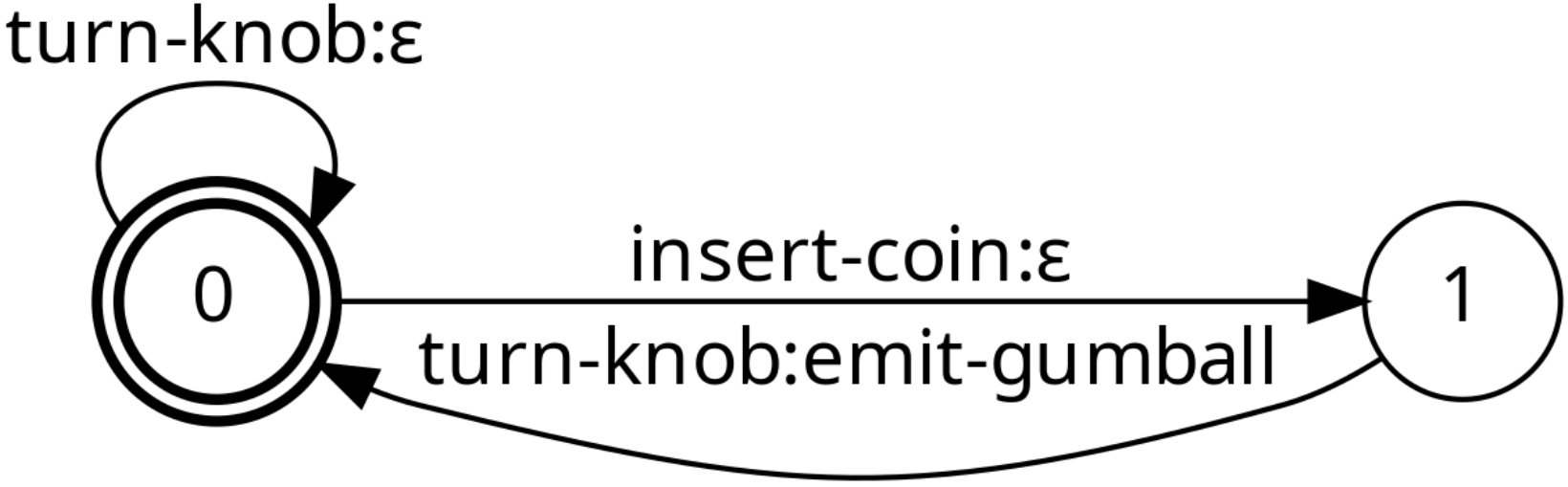
State machines are examples of what computer scientists call *directed graphs*. These are "directed" in the sense that the existence of an arc from state *q* to state *r* does not imply an arc from *r* to *q*. In *state diagrams*, we indicate this directionality using arrows.

(image: credit: Wikimedia Commons)

The humble gumball machine

One familiar example of a state machine—encoded in hardware, rather than software—is the old-fashioned gumball machine. Each state of the gumball machine is associated with actions such as

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


turn-knob:ε
0
insert-coin:ε
1
turn-knob:emit-gumball

Formalization

Sets

A set is an abstract, unordered collection of distinct objects, the *members* of that set. By convention capital Italic letters denote sets and lowercase letters to denote their members. Set membership is indicated with the $\in$ symbol; e.g., $x \in X$ is read "x is a member of X". The empty set is denoted by $\varnothing$.

Subsets

A set X is said to be a *subset* of another set Y just in the case that every member of X is also a member of Y. The subset relationship is indicated with the $\subseteq$ symbol; e.g., $X \subseteq Y$ is read as "X is a subset of Y". Every set is a subset of itself; e.g., $X \subseteq X$.

Union and intersection

- The *union* of two sets, $X \cup Y$, is the set that contains just those elements which are members of X, Y, or both.

$$X \cup Y = \{x \mid x \in X \vee x \in Y\}$$

- The *intersection* of two sets, $X \cap Y$, is the set that contains just those elements which are members of both X and Y.

$$X \cap Y = \{x \mid x \in X \wedge x \in Y\}$$

Strings

Let Σ be an *alphabet* (i.e., a finite set of symbols). A *string* (or *word*) is any finite ordered sequence of symbols such that each symbol is a member of Σ. By convention typewriter text is used to denote strings. The empty string is denoted by ϵ (*epsilon*). String sets are also known as *languages*.

Concatenation and closure

- The *concatenation* of two languages, $X\ Y$, consists of all strings formed by concatenating a string in X with a string in Y.

$$X\ Y = \{xy \mid x \in X \wedge y \in Y\}$$

- The *closure* of a language, X^*, is an infinite language consisting of zero or more "self-concatenations" of X with itself.

$$\begin{aligned} X^* &= \{\epsilon\} \cup X^1 \cup X^2 \cup X^3 \ldots \\ &= \{\epsilon\} \cup X \cup XX \cup XXX \ldots \end{aligned}$$

Regular languages

- The empty language $\varnothing$ is a regular language.
- The empty string language $\{\epsilon\}$ is a regular language.
- If $s \in \Sigma$, then the singleton language $\{s\}$ is a regular language.
- If X is a regular language, then its closure X^* is a regular language.
- If X, Y are regular languages, then:
  - their concatenation XY is a regular language, and
  - their union $X \cup Y$ is a regular language.
- Other languages are not regular languages.

Cross-product

A *pair* or *two-tuple* is a sequence of two elements; e.g., (a, b) is the pair consisting of a then b. The *cross-product* (or *Cartesian product*) of two sets, $X \times Y$, is the set that contains all pairs (x, y) where x is an element of X and y is an element of Y.

$$X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$$

Relations

A (*two-way* or *binary*) *relation* over sets X and Y is a subset of the cross-product $X \times Y$. By convention lowercase Greek letters indicate relations, and the *domain*—set of inputs—and *range*—the set of outputs—are usually provided upon first definition. For example, the "less than" relation might be written $\lambda \subseteq \mathbb{R} \times \mathbb{R} = \{(x, y) \mid x < y\}$ where $\mathbb{R}$ is the set of real numbers.

Functions

A *function* is a relation for which every element of the domain is associated with exactly one element of the range.

Problem

Let $\mathbb{R}$ be the set of real numbers, and $\mathbb{N}$ be the set of natural numbers. Then, are the following relations functions?

- The "less than" relation $\lambda \subseteq \mathbb{R} \times \mathbb{R} = \{(x, y) \mid x < y\}$?
- The "successor" relation $\sigma \subseteq \mathbb{N} \times \mathbb{N} = \{(x, x + 1) \mid x \in \mathbb{N}\}$?

Solution

- λ is not a function because there are an infinitude of real numbers that are greater than any other real number.
- σ is a function because each natural number has just one successor.

N-ary relations

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 $C \times D$.

Application

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

Finite-state acceptors

An *finite-state acceptor* (FSA) is a 5-tuple defined by:

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

Note that, as formalized here, there is exactly one start state but may be multiple final states, and that the start state may also be a final state.

Acceptance

An FSA is said to *accept*, *match*, or *recognize* a string if there exists a path from the initial state to some final state, and the labels of the arcs traversed by that state correspond to the string in question. The set of all strings so accepted are called the FSA's language.

Paths

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

- a state sequence $q_1, q_2, \ldots, q_n \in Q^n$ and a
- 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.

Complete paths

A path is said to be *complete* if

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

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. Then, an FSA accepts string $z \in (\Sigma \cup \{\epsilon\})^*$ if there exists a complete path with string z.

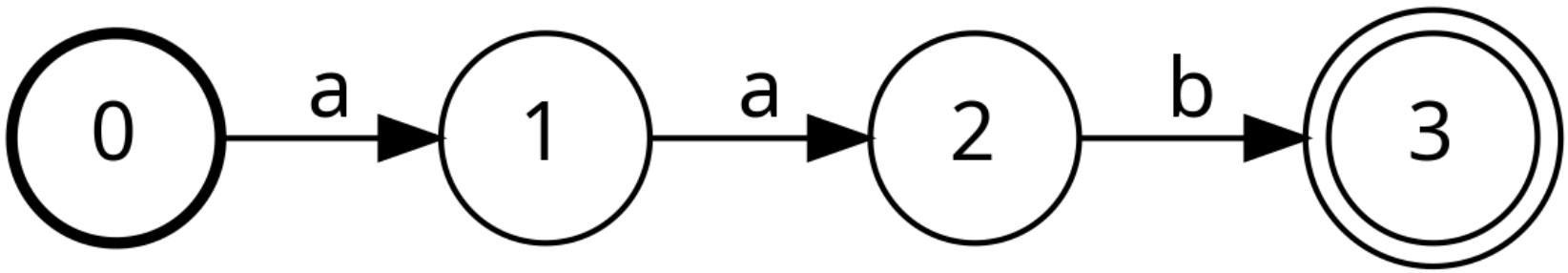
Kleene's theorem

Kleene's theorem holds that any regular language is accepted by an FSA, and any language accepted by an FSA is a regular language. This implies that because regular languages are closed under closure, concatenation, and union, so are FSAs.

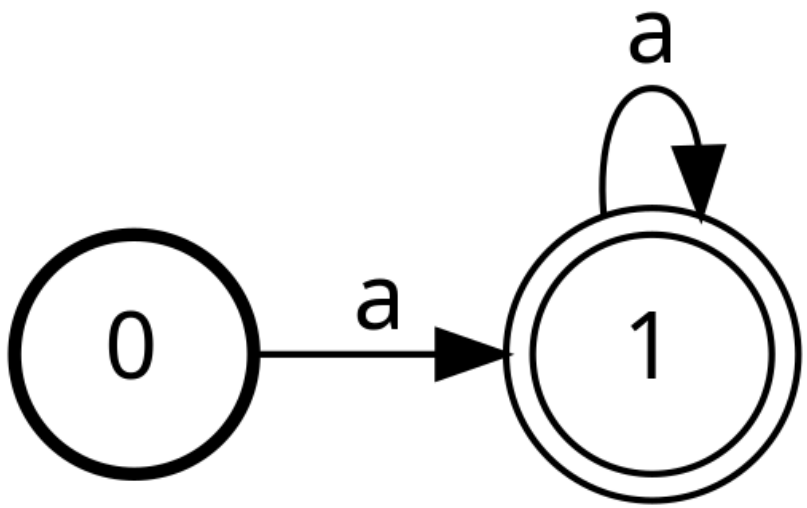
Reading the state diagrams

- States are indicated by circles.
- The initial state is indicated by a bold circle.
- Final states are indicated by double-struck circles.
- Labeled arrows indicate arcs.

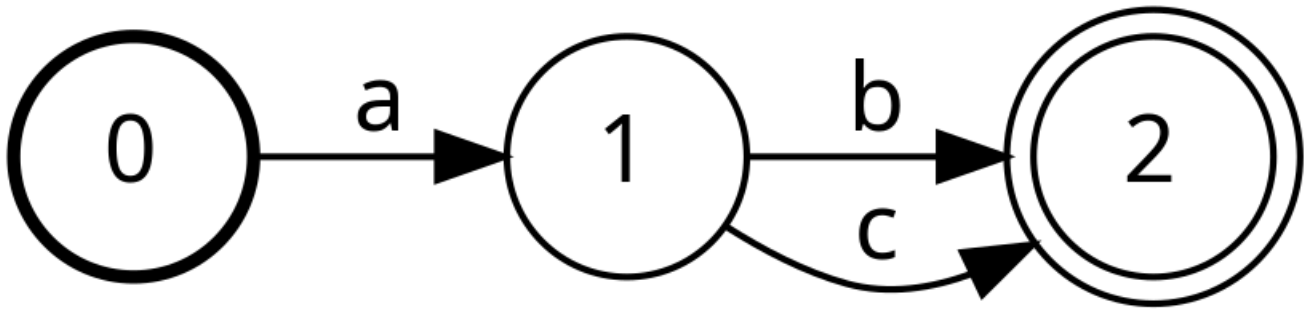
{aab}



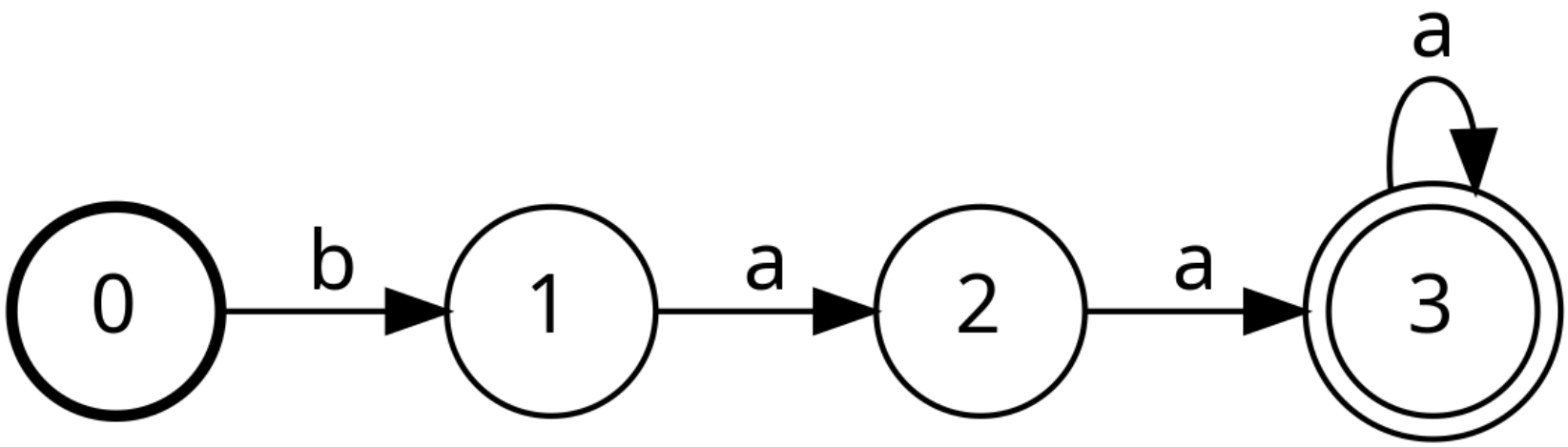
$\{a\}^{+}$



{a}({b} ∪ {c})



$\{ba\}\{a\}^{+}$



The sheep language

- $Q = \{0, 1, 2, 3\}$
- $s = 0$
- $F = \{3\}$
- $\Sigma = \{a, b\}$
- $\delta = \{(0, b, 1), (1, a, 2), (2, a, 3), (3, a, 3)\}$

All about ϵ

The ϵ symbol is a special one which does not match/consume any other symbol. Every ϵ-FSA has an equivalent ϵ-free (or “e-free”) FSA that can be found using the epsilon-removal algorithm (Mohri, 2002).

Coming soon

- Next week we'll introduce separate input and output labels, introducing *finite-state transducers* (FSTs) and the *rational relations* they model.
- In two weeks we'll enrich FSAs and FSTs with weights, giving rise to the *weighted finite-state acceptors* (WFSAs) and *weighted finite-state transducers* (WFSTs), which allow us to assign probabilities to regular languages and rational relations, respectively.

OpenFst and friends

OpenFst (Allauzen et al., 2007)

OpenFst is a open-source C++17 library for weighted finite state transducers developed at Google. Among other things, it is used in:

- Speech recognizers (e.g., Kaldi and many commercial products)
- Speech synthesizers (as part of the “front-end”)
- Input method engines (e.g., mobile text entry systems)

OpenFst design

There are (at least) four layers to OpenFst:

- A C++ template/header library in `<fst/*.h>`
- A C++ “scripting” library in `<fst/script/*.{h,cc}>`
- CLI programs in `/usr/local/bin/*`
- A Python extension module `pywrapfst`

OpenGrm

- Baum-Welch (Gorman et al., 2021; Gorman and Allauzen, 2024): CLI tools and libraries for performing expectation maximization over WFSTs
- NGram (Roark et al., 2012): CLI tools and libraries for building conventional n-gram language models encoded as WFSTs
- Pynini (Gorman, 2016; Gorman and Sproat, 2021): Python extension module for finite-state grammar development
- Thrax (Roark et al., 2012): DSL-based compiler for finite-state grammar development
- SFst (Allauzen and Riley, 2018): CLI tools and libraries for building *stochastic FSTs*

All the OpenGrm tools are built upon, and use the same file formats as, OpenFst.

Speech grammars at Google

Pynini is used extensively at Google for speech-oriented FST grammar development, e.g.:

- Gorman and Sproat (2016) propose an algorithm—implemented in Pynini—which can induce number name grammars from a few-hundred labeled examples.
- Ritchie et al. (2019) describe how Pynini is used to build "unified" verbalization grammars that can be share by both ASR and TTS.
- Ng et al. (2017) constrain a linear-model-based verbalizers with FST covering grammars.
- Zhang et al. (2019) constrain RNN-based verbalizers with FST covering grammars.

More information

openfst.org

opengrm.org

baumwelch.opengrm.org

ngram.opengrm.org

pynini.opengrm.org

thrax.opengrm.org

sfst.opengrm.org

Demo

References I

C. Allauzen and M. Riley. Algorithms for weighted finite automata with failure transitions. In *Proceedings of the 23rd International Conference on Implementation and Application of Automata*, pages 46–58, 2018.

C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: a general and efficient weighted finite-state transducer library. In *Proceedings of the 12th International Conference on Implementation and Application of Automata*, pages 11–23, 2007.

N. Chomsky and G. A. Miller. Introduction to the formal analysis of natural languages. In R. D. Luce, R. R. Bush, and E. Galanter, editors, *Handbook of Mathematical Psychology*, pages 269–321. Wiley, 1963.

E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

References II

K. Gorman. Pynini: a Python library for weighted finite-state grammar compilation. In *ACL Workshop on Statistical NLP and Weighted Automata*, pages 75–80, 2016.

K. Gorman and C. Allauzen. A* shortest string decoding for non-idempotent semirings. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics*, 2024.

K. Gorman and R. Sproat. Minimally supervised number normalization. *Transactions of the Association for Computational Linguistics*, 4:507–519, 2016.

K. Gorman and R. Sproat. *Finite-State Text Processing*. Morgan & Claypool, 2021.

References III

K. Gorman, C. Kirov, B. Roark, and R. Sproat. Structured abbreviation expansion in context. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 995–1005, 2021.

S. C. Kleene. Representations of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, 1956.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

M. Mohri. Generic epsilon-removal and input epsilon-normalization algorithms for weighted transducers. *International Journal of Computer Science*, 13(1):129–143, 2002.

A. H. Ng, K. Gorman, and R. Sproat. Minimally supervised written-to-spoken text normalization. In *IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 665–670, 2017.

References IV

S. Ritchie, R. Sproat, K. Gorman, D. van Esch, C. Schallhart, N. Bampounis, B. Brard, J. F. Mortensen, M. Holt, and E. Mahon. Unified verbalization for speech recognition & synthesis across languages. In *Proceedings of INTERSPEECH*, pages 3530–3534, 2019.

B. Roark, R. Sproat, C. Allauzen, M. Riley, J. Sorensen, and T. Tai. The OpenGrm open-source finite-state grammar software libraries. In *Proceedings of the ACL 2012 System Demonstrations*, pages 61–66, 2012.

K. Thompson. Programming techniques: regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

K. Vijay-Shanker, D. J. Weir, and A. K. Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In *25th Annual Meeting of the Association for Computational Linguistics*, pages 104–111, 1987.

References V

H. Zhang, R. Sproat, A. H. Ng, F. Stahlberg, X. Peng, K. Gorman, and B. Roark. Neural models of text normalization for speech applications. *Computational Linguistics*, 45(2):293–337, 2019.