Takeaway Notes: Finite State Automata

Contents

1 Introduction **1**

2 Basics and Ground Rules **2**
2.1 Building Blocks 2
2.2 The Name of the Game 2

3 Deterministic Finite State Automata **3**
3.1 About that δ 3
3.2 $\hat{\delta}$ Leads to L 4

4 Nondeterministic Finite State Automata **4**
4.1 NFAs and DFAs are Equivalent! 4
4.2 Implications of NFA-DFA Equivalency 5
4.3 The ϵ-Transition 5

5 Important Skills **5**

6 Big Picture Points **6**

1 Introduction

This writeup is meant to provide key takeaways regarding finite state machines or automata. It is largely a distillation of the textbook and online video material, but seen through my individual lens in terms of emphasis and approach. Think of the textbook, the videos, this writeup, and the class meetings as if they were four portholes into the same subject. By exposing yourself to the same material in four different ways, the hope is that you walk away from this in a way that fulfills the objectives of the course.

2 Basics and Ground Rules

2.1 Building Blocks

- An alphabet is a finite set of symbols. Symbols can be anything. Our convention for denoting an alphabet mathematically is Σ.
- We *concatenate* symbols from Σ into *strings*. Strings are finite in length. Empty strings are denoted by ϵ. The length of a string w is denoted by $|w|$.
- Σ^n denotes the set of all strings made up of symbols from Σ whose length is n. $\Sigma^0 = \{\epsilon\}$. Σ^* is the set of *all all all* strings made up of symbols from Σ. Some fun equivalency games:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$$

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$$

$$\Sigma^* = \Sigma^0 \cup \Sigma^+ = \{\epsilon\} \cup \Sigma^+$$

- A language L is any set of strings (strings of finite length, but there may be an infinite number of them) in Σ. More formally, we can say that L is some subset of Σ^*.

2.2 The Name of the Game

We distill our notion of "computation" here to just *recognizing whether some string in Σ belongs to a language L*. The premise behind this distillation is that it "reduces" the problem to something precise and straightforward, with the idea that if something in this realm is already difficult to do, then everything else must be even more difficult than that.

Similarly, although Σ can really consist of any number of symbols, most of the examples we'll see restrict Σ to two symbols. This is done primarily to keep from obligating ourselves to write out tons and tons of state transitions; it turns out that working with two-symbol alphabets is sufficient to convey all of the key ideas in this topic. Larger alphabets merely imply more of everything (states and state transitions). It's a classic case of the mathematical phrase "without loss of generality."

3 Deterministic Finite State Automata

The intuitive notion behind a deterministic finite state automaton or machine (DFA) is pretty straightforward. The trick is remembering how they translate formally, so that theorems can be proven about them.

A DFA A is formalized as a tuple (this "tuple-ization" technique will be used over and over again, so it's worth remembering):

$$A = (Q, \Sigma, \delta, q_0, F)$$

Yep, that's a deterministic finite state machine. The symbols break down like this:

- Q is A's set of states
- Σ is its alphabet
- δ is the big one: it is A's *transition function*. This is effectively the diagram that we draw, but made completely formal and unambiguous
- q_0 is A's start state; hope it's obvious that $q_0 \in Q$
- F is A's accepting states; similarly as above, $F \subset Q$

3.1 About that δ

As stated above, there's a lot more that we can say about δ. To define it further, δ is a function $\delta(q, a)$ where q is a state in Q and a is a symbol in Σ. The value $\delta(q, a)$ is some other state p (also in Q, of course) that represents the new state of the machine if it is currently in state q and receives the symbol a.

As human beings, it's very hard to think about δ in this way, so instead we use either state transition diagrams or transition tables to spell things out.

So "running" a DFA is about computing the value of δ over and over again. We start at q_0 then take the input string one symbol at a time. This gives us $\delta(q_0, a)$. Then we take the next symbol and compute $\delta(\delta(q_0, a), b)$. And so forth. We stop when the latest δ result is a state in F.

This repeated execution leads us to the notion of an *extended transition function*. Instead of a state and a character, this function $\hat{\delta}$ takes a state and a *string*. Its result is the state of the machine after "consuming" the string starting at a given state.

3.2 $\hat{\delta}$ Leads to L

The "extended transition" concept makes it easy for us to formally state if a machine "accepts" some input string w: w is accepted if $\hat{\delta}(q_0, w) \in F$.

Thus, we can now formally define the language L of a DFA $A = Q, \Sigma, \delta, q_0, F$, written as $L(A)$ (the "language of the machine A"):

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \in F\}$$

Set notation! Hope you remember that from past courses.

4 Nondeterministic Finite State Automata

A major extension of the way we can describe finite state automata is to introduce *nondeterminism*. Conceptually, nondeterministic finite state automaton (NFA) can be transition to *multiple* states given the same symbol. This creates the image of having multiple "paths" through the machine for the same string. The NFA accepts a string if a path to an accepting/final state simply *exists*.

The formal definition of an NFA is the same as that of a DFA except for the output of δ. In an NFA, $\delta(q, a)$ is a *set* of states rather than a single one. This set of states is a subset of Q.

Because δ is different for an NFA, then so is $\hat{\delta}$. For an NFA, $\hat{\delta}$ is the *union* of all intermediate state subsets produced by δ for a given string.

And thus, the language that is accepted by a given NFA now becomes:

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

4.1 NFAs and DFAs are Equivalent!

NFAs make finite state machines much easier to define; however, a key finding in the study of these machines is that *a DFA can be constructed from any NFA*. In other words, they can all recognize the same set of languages. In other other words, *NFAs aren't any more powerful than DFAs*.

This is a big claim, and therefore it must be formally proven. The sketch of the proof consists of the following steps:

1. State the algorithm that constructs a DFA out of an NFA (tl;dr the states in the DFA consist of *combinations* of states from the NFA, representing where the machine *could* be after receiving the next symbol in a string)

2. Show that for the machine $D = (Q_D, \Sigma, \delta_D, q_0, F_D)$ that emerges from $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ via this algorithm, $L(D) = L(N)$

3. i.e.,
$$\{w \mid \hat{\delta}_D(q_0, w) \in F_D\} = \{w \mid \hat{\delta}_N(q_0, w) \cap F_N \neq \emptyset\}$$

4.2 Implications of NFA-DFA Equivalency

Key points of this major finding:

- We are free to use NFAs to describe languages because we are assured that there is a DFA out there which will accept the exact same language.
- We like this freedom because, in many cases, NFAs are easier to define. They aren't any more *powerful*, but they are certainly more *convenient*.
- The *cost* of DFA conversion is the number of states: in the worst case, an NFA consisting of n states may blow up into a DFA of 2^n states, due to the way the constructive algorithm works.

4.3 The ϵ-Transition

A significant NFA extension is the inclusion of the empty string ϵ as one of the arguments into the transition function δ. i.e., These kinds of NFAs can change from one state to another *without reading the next symbol in the string.* This is intuitively sensible because NFAs can be in more than one state for the same input string anyway. If this is already true, why wait to read another symbol in order to go to the next state? ϵ transitions simply add to the set of possible states that a machine can be in for a given input string.

5 Important Skills

What should one be able to *do* with this thought framework? Here are some key skillz:

- Define a DFA or NFA that will recognize some language L
- Infer the language L that is recognized by some DFA or NFA
- Prove a statement based on the formal definitions within this topic and/or a specific DFA or NFA tuple

- Have an intuitive sense for what languages *cannot* be recognized by any DFA nor NFA

Note that these correspond to what you are/will be asked to submit as course work for this general topic.

6 Big Picture Points

Things you shouldn't forget even years after taking this course:

- Finite state automata represent the simplest model of computation that has been formally studied.
- As with all models of computation, finite state machines are equivalent to some language (i.e., the language whose strings they accept).
- Nondeterminism adds *convenience* but not *power* to these machines.

And, looking ahead: keep an eye out for what extension/enhancement to the model *does* put it over the top in order to recognize the next level of languages.